How to Use the OSA Guide and Reference

This reference is a detailed technical guide and reference for application programmers creating programs using OS/2 Warp. It gives reference information and code examples to enable you to write source code using OSA functions, classes, methods, and data types.

Before you begin to use this information, it would be helpful to understand how you can:

- Expand the Contents to see all available topics
- Obtain additional information for a highlighted word or phrase
- Use action bar choices
- Use the programming information.

How to Use the Contents

When the Contents window first appears, some topics have a plus (+) sign beside them. The plus sign indicates that additional topics are available.

To expand the Contents if you are using a mouse, click on the plus sign. If you are using the keyboard, use the Up or Down Arrow key to highlight the topic, and press the **Plus** (+) key. For example, **Code Pages** has a plus sign beside it. To see additional topics for that heading, click on the plus sign or highlight that topic and press the **Plus** (+) key.

To view a topic, double-click on the topic (or press the **Up Arrow** or **Down Arrow** key to highlight the topic, and then press the **Enter** key).

How to Obtain Additional Information

After you select a topic, the information for that topic appears in a window. Highlighted words or phrases indicate that additional information is available. You will notice that certain words and phrases are highlighted in green letters, or in white letters on a black background. These are called hypertext terms. If you are using a mouse, double-click on the highlighted word. If you are using a keyboard, press the **Tab** key to move to the highlighted word, and then press the **Enter** key. Additional information then appears in a window.

How to Use Action Bar Choices

Several choices are available for managing information presented in this book. There are three pull-down menus on the action bar: the **Services** menu, the **Options** menu, and the **Help** menu.

The actions that are selectable from the **Services** menu operate on the active window currently displayed on the screen. These actions include the following:

Bookmark

Allows you to set a placeholder so you can retrieve information of interest to you.

When you place a bookmark on a topic, it is added to a list of bookmarks you have previously set. You can view the list, and you can remove one or all bookmarks from the list. If you have not set any bookmarks, the list is empty.

To set a bookmark, do the following:

1. Select a topic from the Contents.
2. When that topic appears, choose the **Bookmark** option from the **Services** pull-down.
3. If you want to change the name used for the bookmark, type the new name in the field.
4. Click on the **Place** radio button (or press the **Up Arrow** or **Down Arrow** key to select it).
5. Click on **OK** (or select it and press **Enter**). The bookmark is then added to the bookmark list.

Search

Allows you to find occurrences of a word or phrase in the current topic, selected topics, or all topics.

You can specify a word or phrase to be searched. You can also limit the search to a set of topics by first marking the topics in the Contents list.

To search for a word or phrase in all topics, do the following:

1. Choose the **Search** option from the **Services** pull-down.
2. Type the word or words to be searched for.
3. Click on **All sections** (or press the **Up Arrow** or **Down Arrow** keys to select it).
4. Click on **Search** (or select it and press **Enter**) to begin the search.
5. The list of topics where the word or phrase appears is displayed.

Print

Allows you to print one or more topics. You can also print a set of topics by first marking the topics in the Contents list.

To print the document Contents list, do the following:

1. Choose **Print** from the **Services** pull-down.
2. Click on **Contents** (or press the **Up Arrow** or **Down Arrow** key to select it).
3. Click on **Print** (or select it and press **Enter**).
4. The Contents list is printed on your printer.

Copy

Allows you to copy a topic that you are viewing to the System Clipboard or to a file that you can edit. You will find this particularly useful for copying syntax definitions and program samples into the application that you are developing.

You can copy a topic that you are viewing in two ways:

- **Copy** copies the topic that you are viewing into the System Clipboard. If you are using a Presentation Manager editor (for example, the System Editor) that copies or cuts (or both) to the System Clipboard, and pastes to the System Clipboard, you can easily add the copied information to your program source module.
- **Copy to file** copies the topic that you are viewing into a temporary file named TEXT.TMP. You can later edit that file by using any editor. You will find TEXT.TMP in the directory where your viewable document resides.

  To copy a topic, do the following:

  1. Expand the Contents list and select a topic.
  2. When the topic appears, choose **Copy to file** from the **Services** pull-down.
  3. The system puts the text pertaining to that topic into the temporary file named TEXT.TMP.

For information on one of the other choices in the **Services** pull-down, highlight the choice and press the **F1** key.

The actions that are selectable from the **Options** menu allow you to change the way your Contents list is displayed. To expand the Contents and show all levels for all topics, choose **Expand all** from the **Options** pull-down. You can also press the **Ctrl-*** keys together. For information on one of the other choices in the **Options** pull-down, highlight the choice and press the **F1** key.

The actions that are selectable from the **Help** menu allow you to select different types of help information. You can also press the **F1** key for help information about the Information Presentation Facility (IPF).

How to Use the Programming Information

This document consists of guide and reference information that provides a detailed description of each function, message, constant, and data type. It provides language-dependent information about the functions which enable the user to generate call statements in the C Language.

Open Scripting Architecture programming information is presented by component, such as classes, instance methods, and class methods; for example:

```
          Contents


+ Notices
+ How to Use the OSA Guide and Reference
  .
  .
  .
+ OSA Event Manager
+ OSA Classes and Methods
```

By clicking on the plus sign beside "OSA Classes and Methods", you see an alphabetic list of the OSA classes. By clicking on the plus sign beside one of the classes, you see an alphabetic list of the methods. Selecting a method takes you directly into the reference information for that method.

Units of reference information are presented in selectable multiple windows or viewports. A viewport is a Presentation Manager window that can be sized, moved, minimized, maximized, or closed. By selecting a unit (in this case, an entry on the Contents list), you will see two windows displayed:

```
Unit Title                Selection Title
```

```
Select an item:

Syntax
Returns
Remarks
Related Methods
Exception Handling
Override Policy
Glossary
```

The window on the left is the primary window. It contains a list of items that are always available to you. The window on the right is the secondary window. It contains a " snapshot" of the unit information. For reference units (that is, method descriptions), this window contains the Method Syntax.

All of the information needed to understand a reference unit (or topic) is readily available to you through the primary window. The information is divided into discrete information groups, and only the appropriate information group appears for the topic that you are viewing.

The information groups for a reference unit (that is, a method description) can include all or some of the following:

- Syntax
- Returns
- Remarks
- Related Methods
- Exception Handling
- Override Policy
- Glossary

This list may vary. Some topics may be omitted when they do not apply.

Information groups are displayed in separate viewports that are stacked in a third window location that overlaps the secondary window. By selecting an item (information group) in the primary window, the item is displayed in the third window location, as follows:

```
Unit Title           Selection       Glossary

Select an item:                    Select a starting
                                   letter of
   .                               glossary terms
   .
   .                               A    N
   .                               B    O
   .                               C    P
Glossary                           .    .
                                   .    .
                                   .    .
                                   M    Z
```

By selecting successive items from the primary window, additional windows are displayed on top of the previous windows displayed in the third window location. For example, in a function description, Parameters and Return Values are items listed in the primary window. When selected, they appear one on top of the other in the third window location. Because of this, you may move the first selected (topmost) window to the left before selecting the next item. This allows simultaneous display of two related pieces of information from the "stack" of windows in the third window location, as follows:

```
Unit Title        Parameters     Return Values

Select an item
   .
   .
   .
Returns
Errors
   .
   .
   .
```

Each window can be individually closed from its system menu. All windows are closed when you close the primary window.

Some secondary windows may have the appearance of a split screen. For example, an illustration may appear in the left half of the window, and scrollable, explanatory information may appear in the right half of the window. Because illustrations may not necessarily fit into the small window size on your screen, you may maximize the secondary window for better readability.

Conventions Used in this Reference

The purpose of this reference is to give information about classes, methods, constants, and data types. It provides information about the methods which enables the user to call functions in the C programming language.

The following information is provided:

- The syntax and parameters for each method.
- The syntax of each data type and structure

Prerequisite Knowledge

This book is intended for application designers and programmers who are familiar with the following:

- C++ Programming Language
- Object-oriented programming concepts
- Object REXX

Programming experience on a multitasking operating system would also be helpful.

Related Publications

The following related publications might be useful when you are creating applications that communicate with other applications through the use of scripts:

- *OSA Event Registry: Standard Suites*
- *Presentation Manager Programming Guide-Advanced Topics*
- *Presentation Manager Programming Guide-The Basics*
- *Presentation Manager Programming Reference*
- *Object REXX Programming Guide for OS/2*
- *Object REXX Reference for OS/2*

Conventions Used in This Book

The following notation and document conventions are used in this book.

Notation Conventions

The following notation conventions are used:

NULL	The term NULL applied to a parameter is used to indicate the presence of a pointer parameter that has no value.
NULLHANDLE	The term NULLHANDLE applied to a parameter is used to indicate the presence of the handle parameter, but with no value.
Implicit Pointer	If no entry for a data type "Pxxxxxxx" exists in Data Types, then it is implicitly a pointer to the data type "xxxxxxx". See Implicit Pointer Data Types for more information about implicit pointers.

Documentation Conventions

Throughout this library of documents, the following conventions distinguish the different elements of text:

bold	Menu bar and menu choices, names of keys, push buttons, check boxes, radio buttons, and field names.
plain text	Function names, structure names, data types names, enumerated types, and constant names.
italics	Parameters, structure fields, titles of documents, and first occurrences of words with special meanings.
CAPITALS	File names and error message text.
`monospace`	Programming examples and user input at the command line prompt or in an entry field.

Big and Little Endian Considerations

OSA events use a big endian data format. Although OS/2 is a little endian platform, it uses big endian data format for OSA events in order to maintain compatibility and cross-platform event distribution.

The OS/2 header files have been modified to use big endian byte ordering for all OSA-related data, including descriptor types and keywords. For example, the OS/2 definition for the typeChar descriptor type is 'TXET' instead of 'TEXT'.

In general, all data placed into an AEDesc structure must be in big endian format; however, there is an exception to this rule for integer data types. These types are very pervasive, and it is important to minimize the overhead of having to constantly convert integers back and forth from little to big endian. To alleviate this problem, all platforms provide automatic coercion routines for integer descriptor types. Integers can be sent in OSA events in the platform's native format, big or little endian. They are coerced to the proper byte order, if necessary, by coercion routines provided in the OSA Event Manager.

The OS/2 descriptor type definition for typeLong, for example, is 'long' (it has not been reversed). When this descriptor is extracted from an OSA event, it is coerced, if necessary. In this case, 'gnol' is placed into an OS/2 AEDesc structure.

Consider an integer received in an OSA event from a big endian platform. It has 'long' in the AEDesc structure. When it is extracted from the event with an API, such as AEGetNthDesc, the *desiredType* parameter is 'gnol', since the OS/2 definition for typeLong is used. The OSA Event Manager invokes the 'long' to 'gnol' coercion handler and returns the integer in little endian format. In this example, if the OSA event had been received from a little endian platform, no coercion would be required.

The OSA integer data types that are maintained in little endian format are:

Descriptor Type	Data Type	OS/2 Definition
typeLongInteger	signed long	long
typeShortInteger	signed short	shor
typeInteger	signed long	long
typeSMInt	signed short	shor
typeMagnitude	unsigned long	magn

Programming Considerations

This section covers aspects of Open Scripting Architecture development that programmers need to consider before creating PM applications

or OpenDoc part handlers.

OSA Event Registry

This book contains references to a document called *OSA Event Registry: Standard Suites for OS/2* . This document is not available at this time. Eventually, it will be a cross-platform compatible version of the *Apple Event Registry: Standard Suites* .

Until the new document is available, you must continue to use the *Apple Event Registry: Standard Suites* with the following modifications to the event definitions in OS/2:

- The typeAlias descriptor is replaced by typeOS2FileName
- The typeIntlText descriptor is replaced by typeOS2IText

These changes apply to the core, required, text, and table suites as well as the miscellaneous standards. The types are defined in the ODOSACOM.H file. This file also defines the PlainITextData structure used in descriptors of type typeOS2IText.

Defining new aete terms

In order to avoid conflicts with existing suite definitions, uppercase letters should always be used when defining four character codes for new OSA terminology. This minimizes conflicts with Apple defined terms.

Object REXX restrictions

The current release of Object REXX supports compilation, get source, and convenience. It does not support coercion, AE sending, recording, dialects, or event handling.

Required OSA events

OS/2 does not send any of the required events (open application, open documents, or print documents) to an application when it is launched. These events may be sent, however, by a scripting component or another application.

User Interaction Level

OS/2 does not provide complete support for user interaction. While the client can set the interaction level in events that it sends, the server cannot use the AESetInteractionAllowed or AEInteractWithUser functions.

Because this release only supports local events sent on one machine and does not support sending events to a remote machine, you should assume that kAEInteractWithLocal is in effect.

Double-Byte Character Set (DBCS)

Throughout this publication, you will see references to specific values for character strings. The values are for single-byte character set (SBCS). If you use the double-byte character set (DBCS), note that one DBCS character equals two SBCS characters.

About Component Integration Laboratories

OpenDoc is presented and maintained through a nonprofit organization devoted to promoting cross-platform standards, architectures, and protocols in a vendor-independent fashion. This organization, Component Integration Laboratories (CI Labs), is composed of a number of platform and application vendors with a common interest in solving OpenDoc issues and promoting interoperability.

CI Labs supports several levels of participation through different membership categories. If you are interested in shaping the future direction of component software, or if you simply need to be kept abreast of the latest developments, you can become a member. For an information packet, send your mailing address to:

Component Integration Laboratories
PO Box 61747
Sunnyvale, CA 94088-1747

Telephone:	408-864-0300
FAX:	408-864-0380
Internet:	cilabs@cilabs.org
World Wide Web:	http://www.cilabs.org

Interapplication Communication

This chapter describes the interapplication communication (IAC) architecture for the OS/2 system, summarizes how your application can take advantage of it, and tells you where, in this book, to find the information you need to perform specific tasks. It also introduces the OSA and describes how to make your application scriptable-that is, capable of responding to OSA events sent to it by a scripting component.

Overview of Interapplication Communication

The *interapplication communication (IAC)* architecture provides a standard and extensible mechanism for communication among OS/2 applications. The IAC architecture makes it possible for your application to do the following:

- Provide automated copy and paste operations between your application and other applications
- Be manipulated by means of scripts
- Send and respond to OSA events

The most important requirement for high-level communication among all applications is a common vocabulary of events. High-level events that conform to this protocol are called *OSA events* .

The vocabulary of publicly available OSA events is published in the *OSA Event Registry: Standard Suites* , which defines the standard OSA events that developers have created for use by all applications. To ensure that your application can communicate at a high level with other applications that support OSA events now and in the future, you should support the standard OSA events that are appropriate for your application.

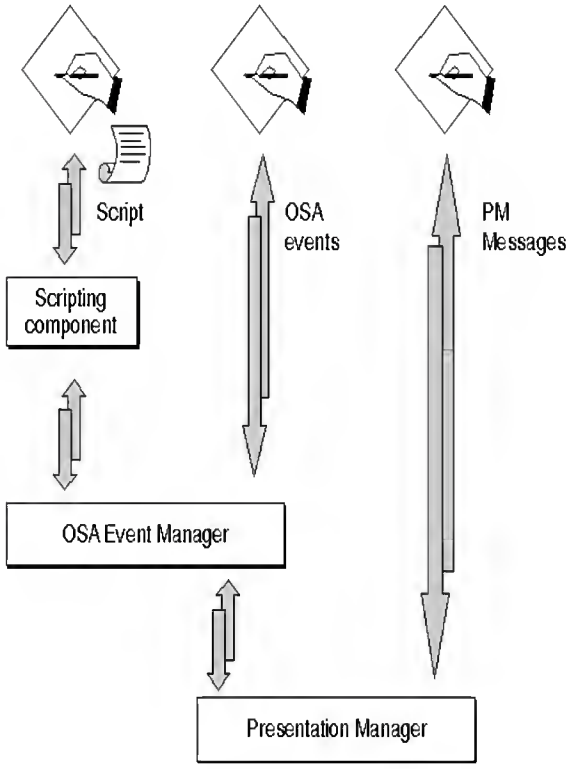
The IAC architecture comprises the following parts:

- *Open Scripting Architecture (OSA)* provides a mechanism that allows users to control multiple applications by means of scripts, or sets of instructions, written in a variety of scripting languages. Each scripting language has a corresponding scripting component that is managed by the Component Manager. When a user executes a script, the scripting component sends OSA events to one or more applications to perform the actions the script describes.
- *OSA Event Manager* allows applications to send and respond to OSA events.
- *Presentation Manager* allows applications to send and respond to high-level events other than OSA events.

The following figure shows the primary relationships among these parts. The managers and components toward the top of the figure rely on the managers beneath them. Scripting components manipulate and execute scripts with the aid of the OSA Event Manager. The OSA Event Manager in turn relies on the Presentation Manager to send OSA events.

The following figure also shows the three principal means of communication provided by the IAC architecture. In addition to using scripting

components to send OSA events on their behalf, applications can use the OSA Event Manager directly to send OSA events to other applications. All applications can use the OSA Event Manager to respond appropriately to OSA events, whether they are sent by a scripting component or other applications.



The three forms of IAC shown in the previous figure can be summarized as follows:

- *Scripting* . The OSA includes the OSA Event Manager, the OSA events defined by the *OSA Event Registry: Standard Suite*s , and the routines supported by s*cripting component*s , which applications can use via the Component Manager to execute scripts. Script-editing applications such as Script Editor (not shown in the previous figure) allow users to manipulate and execute scripts.

  Each scripting language has a corresponding scripting component that can execute scripts written in that language. Scripting components typically implement a text-based scripting language based on OSA events. For example, the *Object REXX component* implements *Object REXX* , the standard user scripting language defined by IBM. When the Object REXX component executes a script, it performs the actions described in the script, including sending OSA events to applications when necessary.

  Supporting Object REXX and Other Scripting Languages describes how the OSA makes it possible for your application to:

  - Provide human-language equivalents to OSA event codes so that scripting components can send your application the appropriate OSA events during script execution
  - Allow users to record their actions in the form of a script when using a scripting component which supports recording.
  - Manipulate and execute scripts

- *Sending and Responding to OSA Event*s . Your application can send OSA events directly to other applications to request services or information or to provide information. To support Object REXX and most other scripting languages based on OSA, your application must be able to respond to OSA events. Sending and Responding to OSA Events describes how applications can send and respond to OSA events with the aid of the OSA Event Manager.

All forms of IAC are based on the premise that applications cooperate with each other. Both the application sending an event and the application receiving it must agree on the protocol for communication. You can ensure effective communication between your application and other OS/2 applications by supporting the standard OSA events defined in the *OSA Event Registry: Standard Suite*s .

Sending and Responding to OSA Events

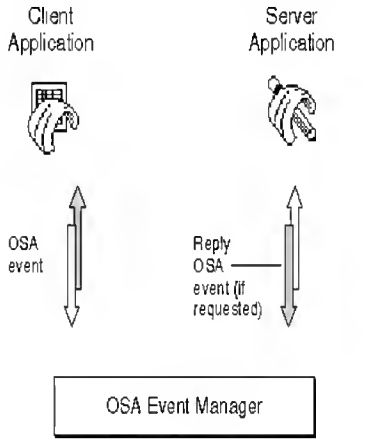
The OSA Event Manager uses the Presentation Manager to send OSA events between applications on the same computer.

Applications typically use OSA events to request services and information from other applications or to provide services and information in

response to such requests. For example, any application can use the Get Data OSA event to request that your application locate and return a particular set of data, such as a table. If your application supports the Get Data event, it should be able to recognize the event and respond by locating the requested data and returning a copy of the data to the application that requested it.

Communication between two applications that support OSA events is initiated by a *client application* , which sends an OSA event to request a service or information. For example, a client application might request services such as printing specific files, checking the spelling of a list of words, or performing a numeric calculation; or it might request information, such as one customer's address or a list of names and addresses of all customers living in Florida. The application providing the service or the requested information is called a *server application* .

The following figure shows the relationships between a client application, the OSA Event Manager, and a server application. The client application uses OSA Event Manager routines to create and send the OSA event, and the server application uses OSA Event Manager routines to interpret the OSA event and respond appropriately. If the client application requests that the server application adds information to a reply OSA event, the server application adds the information which the OSA Event Manager returns to the client application.



If an OSA event is one of the standard events defined in the *OSA Event Registry: Standard Suite*s , the client application can construct the event and the server application can interpret it according to the standard definition for that event. To ensure that your application can respond to OSA events sent by other applications, you should support the standard OSA events that are appropriate for your application.

Standard OSA Events

The current edition of *OSA Event Registry: Standard Suite*s defines the standard s*uite*s of OSA events, which are groups of related events that are usually implemented together. The OSA Event Registrar maintains the *OSA Event Registry: Standard Suite*s and other information about the ongoing development of OSA event suites.

The standard suites include the following:

- The *Required suite* consists of four OSA events. These events are Open Application, Open Documents, Print Documents, and Quit Application. Your application must support the required OSA events as described in Responding to OSA Events.
- The *Core suite* consists of the basic OSA events, including Get Data, Set Data, Move, Delete, and Save, that nearly all applications use to communicate. You should support the OSA events in the Core suite that make sense for your application.
- A *functional-area suite* consists of a group of OSA events that support a related functional area. Functional-area suites include the Text suite and the Database suite. You can decide which functional-area suites to support according to which features your application provides. For example, most word-processing applications should support the Text suite, and most database applications should support the Database suite.

Implementing OSA Events

You do not need to implement all OSA events at once. You should begin by supporting the required OSA events, and then add support for the core events and the functional-area events as appropriate for your application.

If necessary, you can extend the definitions of the standard OSA events to suit specific capabilities of your application. You can also define your own custom OSA events. However, only those applications that choose to support your custom OSA events explicitly will be able to make use of them. If all applications communicated solely by means of custom OSA events, every application would have to support all other

applications' custom events. Instead of creating custom OSA events, try to use the standard OSA events and extend their definitions as necessary.

OSA events describe actions to be performed by the applications that receive them. In addition to a vocabulary of actions, or "verbs," effective communication between applications requires a method of referring to windows, data (such as words or graphic elements), files, folders, volumes, and other items on which actions can be performed. The OSA Event Manager provides a method for specifying structured names, or "noun phrases," that applications can use to describe the objects on which OSA events act.

OSA Events Objects

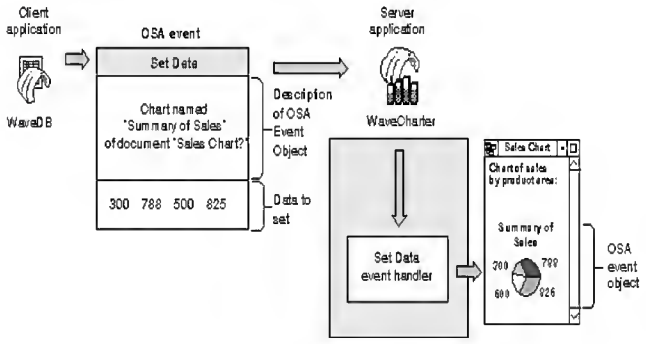
The *OSA Event Registry: Standard Suite*s includes definitions for *OSA event object classes* , which are simply names for objects that can be acted upon by each kind of OSA event. Applications use these definitions and OSA Event Manager routines to create complex descriptions of almost any discrete item in another application or its documents. For example, an application could use OSA Event Manager routines and standard object class definitions to construct a Get Data event that requests "the most recent invoice to John Chapman in the Invoices database on the Archives server" and send the event to the appropriate application across the network.

An *OSA event object* is any item supported by an application, such as a word, paragraph, shape, or document, that can be described in an OSA event. In the example just given, the specified invoice, the Invoices database, and the Archives server are nested OSA event objects. Nearly any item that a user can differentiate and manipulate on an OSA-compliant system can be described as an OSA event object of a specified object class nested within other OSA event objects. When handling an OSA event that includes such a description, an application must locate the specified OSA event object and perform the requested action on it.

Most of the standard OSA events defined in the *OSA Event Registry: Standard Suite*s require your application to recognize specific OSA event object classes. Support for the standard OSA events, including OSA event object classes, allows your application to respond to requests for services or information from any other application or process.

Handling OSA Events

The following figure shows a common OSA event from the Core suite, the Set Data event. The WaveDB application is the client; it sends a Set Data event to the WaveCharter application. This event requests that WaveCharter use some new sales figures generated by WaveDB to update the data for the chart named "Summary of Sales" in the document named "Sales Chart." The OSA event contains information that identifies an action-setting data-and a description of the OSA event object on which to perform the action-"the chart named Summary of Sales in the document named Sales Chart." The OSA event also includes the new data for the chart.



To respond appropriately, the WaveCharter application in the previous figure can use the OSA Event Manager to determine what kind of OSA event has been sent and pass the event to the appropriate OSA event handler. An *OSA event handler* is an application-defined function that extracts pertinent data from an OSA event, performs the requested action, and returns a result. In this case, the Set Data event handler must locate an OSA event object-that is, the specified chart in the specified document-and change the data displayed in the chart as requested.

The OSA Event Manager provides routines that a server application can use in its OSA event handlers to take apart an OSA event and examine its contents. The WaveCharter application in the previous figure can interpret the contents of the Set Data OSA event according to the definition of that event in the *OSA Event Registry: Standard Suite*s . The Set Data event handler uses both OSA Event Manager routines and the WaveCharter application's own routines to locate the chart and make the requested change.

The OSA Event Manager also provides routines that a client application can use to construct and send an OSA event. However, the most important requirement for applications that support IAC is the ability to respond to OSA events, because this ability is essential for an application that users can control through scripts. The next section describes how you can use OSA events to support scripting in your application.

OSA Events provides an overview of OSA events and describes how you can use the OSA Event Manager to implement OSA events in your

application. Responding to OSA Events, Creating and Sending OSA Events, Resolving and Creating Object Specifier Records, and Recording OSA Events provide detailed information about the OSA Event Manager.

Supporting Object REXX and Other Scripting Languages

A *script* is any collection of data that, when executed by the appropriate program, causes a corresponding action or series of actions. For example, some database, telecommunications, and page-layout applications allow users to automate repetitive or conditional tasks by means of scripts written in proprietary scripting languages.

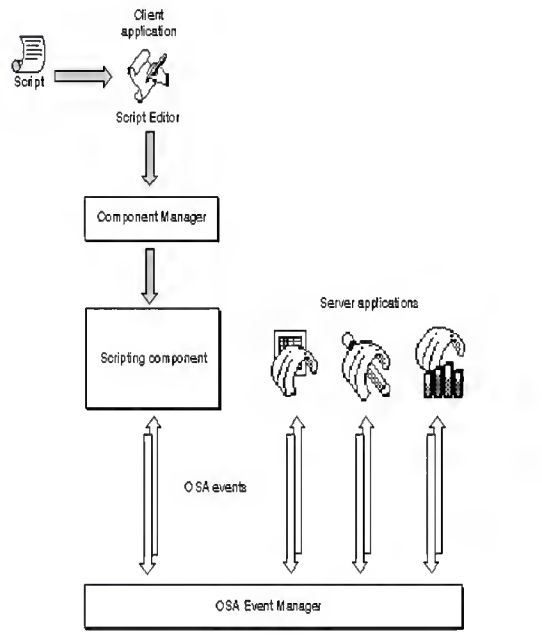
The Open Scripting Architecture (OSA) provides a standard mechanism that allows users to control multiple applications with scripts written in a variety of scripting languages. Each scripting language has a corresponding scripting component. When a scripting component executes a script, it performs the actions described in the script, including sending OSA events to applications if necessary.

The OSA comprises the following parts:

- The OSA Event Manager allows applications to respond to OSA events sent by scripting components (see the previous section, Sending and Responding to OSA Events).
- The *OSA Event Registry: Standard Suite*s defines the standard vocabulary of OSA events.
- The standard scripting component data structures, routines, and resources allow applications to interact with any scripting component.
- The Object REXX component implements the Object REXX scripting language.

The *Object REXX component* , which implements the *Object REXX scripting language* , is the implementation of the OSA provided by IBM.

The following figure shows the relationships among some of these parts. The client application in the following figure is Script Editor, an application provided by IBM, that allows users to record, edit, and execute scripts. The client application could also be any other application that uses the standard scripting component routines to execute scripts. Script Editor uses the Component Manager to open a connection with the scripting component that created the script to be executed.



Scripts can be stored in applications and documents as well as in distinct script files. The Script Editor allows users to execute scripts stored in *script file*s .

During script execution, scripting components perform actions described in the script, using the OSA Event Manager to send OSA events when necessary. The server applications shown in the previous figure use the OSA Event Manager to examine the contents of the OSA events they receive and to respond appropriately. A server application always responds to the same OSA event in the same way, regardless of whether the event is sent by a scripting component or directly by a client application.

You can take advantage of the OSA in three ways:

- You can make your application sc*riptable* , or capable of responding to OSA events sent to it by a scripting component. An application is scriptable if it does the following:

    - Responds to the appropriate standard OSA events. See the previous section, Sending and Responding to OSA Events.

    - Provides an OSA event terminology extension (**aete**) resource that describes which OSA events your application supports and the corresponding human-language terminology for use in scripts. The **aete** resource allows scripting components to interpret scripts correctly and send the appropriate OSA events to your application during script execution. By executing scripts, users of scriptable applications can perform almost any task that they would otherwise perform by choosing menu commands, typing, and so on. Users can also execute scripts to perform many tasks that might otherwise be difficult to accomplish, especially repetitive or conditional tasks that involve multiple applications.

- You can make your application *recordable* -that is, capable of sending OSA events to itself in response to user actions such as choosing a menu command or changing the contents of a document. After a user has turned on recording for a particular scripting component, the scripting component receives copies of all subsequent OSA events and records them in the form of a script.

- You can have your application manipulate and execute scripts with the aid of a scripting component. To do so, your application must:

    - Use the Component Manager to open a connection with the appropriate component

    - Use the standard scripting component routines to record, edit, compile, save, load, or execute scripts when necessary

    Users of applications that execute scripts can modify the applications' behavior by editing the scripts. For example, a user of an invoice program might be able to write a script that checks and if necessary updates customer information in a separate database application each time the user posts an invoice.

The sections that follow describe these three kinds of scripting capabilities in more detail. Introduction to Scripting provides an overview of the way scripting components work and how you can implement support for scripting in your application.

Scriptable Applications

If your application can respond to standard OSA events sent by other applications, it can also respond to the same OSA events sent by a scripting component. Before executing a script that controls your application, a scripting component must associate the human-language terms used in the script with specific OSA event codes supported by your application. Scriptable applications provide this information in an OSA event terminology extension (**aete**) resource.

Because scripting components can obtain information from **aete** resources about the nature of different applications' support for OSA events, a single script can describe complex tasks performed cooperatively by several specialized applications. For example, a user can execute an Object REXX script to locate all records in a database with specific characteristics, update a series of charts based on those records, import the charts into a page-layout document, and send the document to a remote computer on the network via electronic mail.

When a user executes such a script, the Object REXX component attempts to perform the actions the script describes, including sending OSA events to various applications when necessary. To map human-language terms used in the script to the corresponding OSA events supported by each application, the Object REXX component looks up the terms in the applications' **aete** resources. Each human-language term specified by an application's **aete** resource has a corresponding OSA event code. After the Object REXX component has identified the OSA event codes for the terms used in a script, it can create and send the OSA events that perform the actions described in the script.

To respond appropriately to the OSA events sent to it by the Object REXX component, the database application in this example must be able to locate records with specific characteristics so that it can identify and return the requested data. The other applications involved must support OSA events that perform the other actions described in the script.

One line in such a script might be a statement like this:

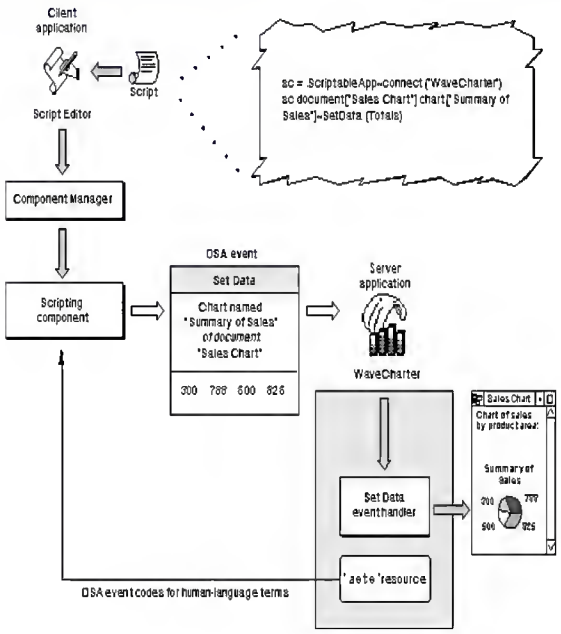
```
sc [document["Sales Chart"] chart["Summary of Sales"][~SetData(Totals)
```

In this statement, the word Totals is a variable that has been set earlier in the same script to the value of the new data generated by a database application. The statement causes the Object REXX component to send a Set Data event updating the chart named "Summary of Sales". The following figure shows how the Object REXX component would execute this statement. (The figure in section Handling OSA Events shows a database application that sends a similar Set Data event directly.)

To interpret the terms in this script statement correctly, the Object REXX component must be able to look them up in the WaveCharter application's **aete** resource, which maps those terms to the corresponding codes for OSA events, object classes, and so on used by the OSA Event Manager. The Object REXX component can then create and send the Set Data event to WaveCharter.

When it receives the Set Data event, the WaveCharter application uses the OSA Event Manager to determine what kind of OSA event has been sent and to pass the event to WaveCharter's handler for that event, which in turn locates the chart and changes its data as requested.

Introduction to Scripting describes how the **aete** resource works. OSA Event Terminology Resources describes how to define terminology for use by the Object REXX component and how to create an **aete** resource.



Recordable Applications

If you decide to make your application scriptable, you can also make it recordable, allowing users to record their actions in your application in the form of a script. Even users with little or no knowledge of a particular scripting language can record their actions in recordable applications in the form of a script. More knowledgeable users can record scripts and then edit or combine them as desired.

Applications generally have two parts: the code that implements the application's user interface and the code that actually performs the work of the application when the user manipulates the interface. To make your application fully recordable, you should separate these two parts of your application, using OSA events to connect user actions with the work your application performs.

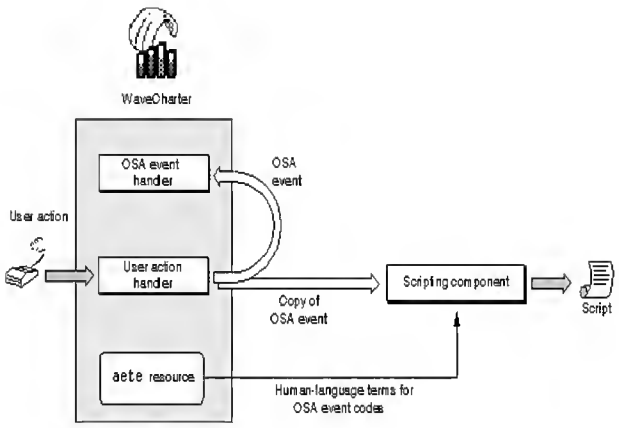
Any significant user action within a recordable application should generate OSA events which a scripting component can record as statements in a script. For example, when a user chooses **New** from the **File** menu, a recordable application sends itself a Create Element event, and the application's handler for that event creates the new document. Implementing OSA events in this manner is called *factoring* your application. A factored application acts as both the client and the server application for the OSA events it sends to itself.

In general, a recordable application should generate OSA events for any user action that could be reversed by the **Undo** command. A recordable application can usually handle a greater variety of OSA events than it can record, because it must record the same action the same way every time even though OSA events might be able to trigger that action in several different ways.

A *recordable event* is any OSA event that any recordable application sends to itself while recording is turned on for the local computer (with the exception of events that the application indicates it does not want to be recorded). After a user turns on recording from the Script Editor application, the OSA Event Manager sends copies of all recordable events to the Script Editor. A scripting component previously selected by the user handles each copied event for the Script Editor by translating the event into the scripting component's scripting language and recording the translation as part of a script. When a scripting component executes a recorded script, it sends the corresponding OSA events to the applications in which they were recorded.

The following figure illustrates how OSA event recording works. The user performs a significant action (such as choosing **New** from the **File** menu), and the WaveCharter application sends itself an OSA event to perform the task associated with that action. If recording is turned on, the OSA Event Manager automatically sends a copy of each recordable OSA event to the application (for example, Script Editor) that initiated recording. The scripting component handles the copy of each recordable event by translating it and recording it as part of a script. To translate each OSA event correctly, the scripting component must first check what equivalent human-language terminology the WaveCharter application uses for that OSA event. The scripting component then records the equivalent statement in the script.

Recording OSA Events describes the OSA Event Manager's recording mechanism in more detail and explains how to use OSA events to factor your application.



Applications That Manipulate and Execute Scripts

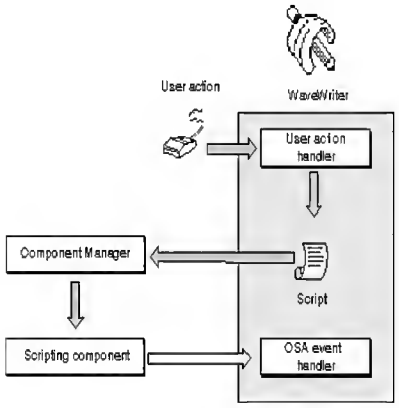
Scripts can be stored either as separate files with their own icons or within documents. Your application can store and execute scripts regardless of whether it is scriptable or recordable. If your application is scriptable, however, it can execute scripts that control its own behavior, thus acting as both the client application and the server application for the corresponding OSA events.

Your application can establish a connection with any scripting component that is registered with the Component Manager on the same computer. Each scripting component can manipulate and execute scripts written in the corresponding scripting language when your application calls the standard scripting component routines.

You can use the standard scripting component routines to do the following:

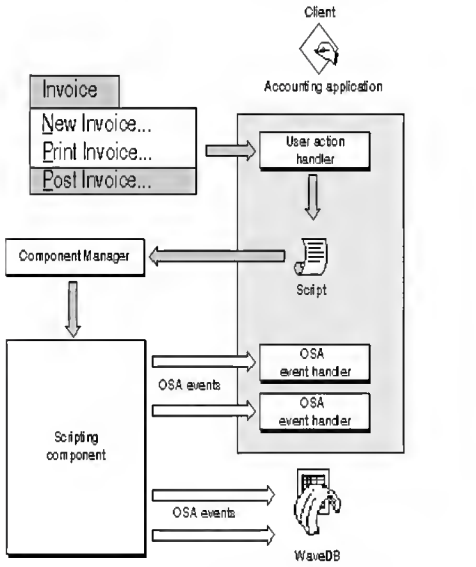
- Get a handle to a script so you can save the script in a script file
- Manipulate scripts associated with any part of your application or its documents, including both OSA event objects and other objects defined by the application
- Let users record and edit scripts
- Compile and execute scripts

The following figure shows how an application might execute a script that controls its own behavior. The appropriate user action handler executes the script in response to a user action, which can be almost anything: choosing a menu command, clicking a button, tabbing from one table cell to another, and so on. The script might consist of a single statement that describes some default action, such as saving or printing, or a series of statements that describe a series of tasks, such as setting default preferences or styles. The following figure shows a script that corresponds to a single OSA event, but the script could just as easily correspond to a whole series of OSA events. If your application allows users to modify such a script, they can modify the behavior of your application to suit their needs.



Your application can associate a script with any OSA event object or application-defined object and execute the script when that object is manipulated in some way. The script can describe actions to be taken by your application, as in the previous figure, or actions to be taken by several applications. For example, a user of a word-processing application might attach a script to a specific word so that the application executes the script whenever that word is double-clicked. Such a script could trigger OSA events that look up and display related information from a separate document, play a video, perform a calculation, play a voice annotation, and so on.

The following figure shows one way that a script can be used to control two or more applications. When a user chooses the **Post Invoice** command in the accounting application, the user action handler for that menu command executes a default script for posting an invoice. That script might describe actions such as saving the invoice, updating the sales journal, and so on. The scripting component sends OSA events to the accounting application to perform these actions.



The accounting application also allows users to open the default invoice-posting script in the Script Editor and modify it so that additional actions are performed when it is executed. For example, as shown in the previous figure, the script could instruct the WaveDB application to update a database of customer information in addition to performing the default posting actions. In this case, the scripting component sends OSA events to both the accounting application and WaveDB to carry out all the actions described by the script.

There is no limit to the actions such a script can describe. In addition to sending the OSA events shown in the previous figure, the invoice-posting script could be used to trigger OSA events that cause other applications to perform a credit check, send the invoice to the customer by electronic mail, forward inventory information to a remote server on the network, and so on.

Scripting Components describes how your application can use the standard scripting component routines to manipulate and execute its own scripts and allow users to modify those scripts.

OSA Events

This chapter introduces OSA events and the OSA Event Manager. Later chapters describe how your application can use the OSA Event Manager to respond to and send OSA events, locate OSA event objects, and record OSA events.

The *OSA Event Registry: Standard Suite*s defines both the actions performed by the standard OSA events, or "verbs," and the standard OSA event object classes, which can be used to create "noun phrases" describing objects on which OSA events act. If your application uses the OSA Event Manager to respond to some of these standard OSA events, you can make it scriptable-that is, capable of responding to scripts written in a scripting language, such as Object REXX. In addition, your application can use the OSA Event Manager to create and send OSA events and to allow user actions in your application to be recorded as OSA events.

This chapter begins by describing OSA events and some of the data structures they contain. The rest of the chapter introduces the use of the OSA Event Manager to:

- Respond to OSA events
- Send OSA events to request services or information
- Work with object specifier records
- Classify OSA event objects
- Locate OSA event objects

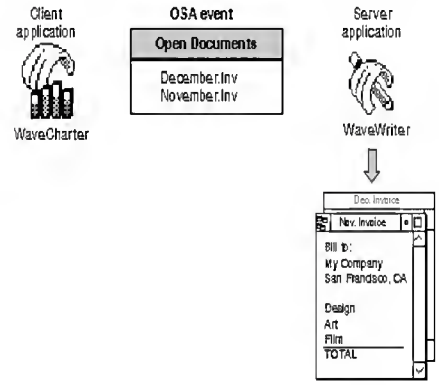
Finally, this chapter summarizes the tasks you can perform with the OSA Event Manager and explains where to locate information you need to perform those tasks.

About OSA Events

The OSA Event Manager uses the services of the Presentation Manager to send *OSA event*s between applications on the same computer.

Applications typically use OSA events to request services and information from other applications or to provide services and information in response to such requests. Communication between two applications that support OSA events is initiated by a *client application* , which sends an OSA event to request a service or information. The application providing the service or the requested information is called a s*erver application* . An application can also send OSA events to itself, thus acting as both client and server.

The following figure shows a common OSA event, the Open Documents event. The WaveCharter application is the client; it requests that the WaveWriter application open the documents named "DECEMBER.INV" and "NOVEMBER.INV." The WaveWriter application responds to the Workplace Shell's request by opening windows for the specified documents.



The WaveCharter application is considered the client application for the Open Documents event shown in the previous figure because the WaveCharter application initiates the request for a service. The WaveCharter application can also be considered the source application for the same Open Documents event. A so*urce application* for an OSA event is one that sends an OSA event to another application or to itself. Similarly, the WaveWriter application can be described as either the server application or the target application for the Open Documents event shown in the previous figure. A *target application* for an OSA event is the one addressed to receive the OSA event. The terms client application and source application are not always synonymous, nor are the terms server application and target application. Typically, an OSA event client application sends an OSA event requesting a service to an OSA event server application; in this case, the server application is the target application for the OSA event. A server application may return information to the client in a reply OSA event-in which case, the client application is the target application for the reply.

To perform the requested service-that is, to open the specified documents-the WaveWriter application shown in the previous figure first uses the OSA Event Manager to identify the event (the Open Documents event) and to dispatch the event to WaveWriter's handler for that OSA event. An *OSA event handler* is an application-defined function that extracts pertinent data from an OSA event, performs the requested action, and (usually) returns a result. In this case, WaveWriter's Open Documents event handler examines the OSA event to determine which documents to open (DECEMBER.INV and NOVEMBER.INV), then opens them as requested.

To identify OSA events and respond appropriately, every application can rely on a vocabulary of standard OSA events that developers have established for all applications to use. These events are defined in the *OSA Event Registry: Standard Suite*s . The standard s*uite*s , or groups of related OSA events that are usually implemented together, include the Required suite, the Core suite, and functional-area suites such as the Text suite and the Database suite. To function as a server application, your application should be able to respond to all the OSA events in the Required suite and any of those in the Core and functional-area suites that it is likely to receive. For example, most word-processing applications should be capable of responding to the OSA events in the Text suite, and most database applications should be capable of responding to those in the Database suite.

If necessary, you can extend the definitions of the standard OSA events to match specific capabilities of your application. You can also define your own custom OSA events; however, before defining custom events, you should check with CI Labs to find out whether you can adapt existing OSA event definitions or definitions still under development to the needs of your application.

By supporting the standard OSA events in your application, you ensure that your application can communicate effectively with other applications that also support them. Instead of supporting many different custom events for a limited number of applications, you can support a relatively small number of standard OSA events that can be used by any number of applications.

You can begin supporting OSA events by making your application a reliable server application: first for the required OSA events, then for the core and functional-area OSA events as appropriate. Once your application can respond to the appropriate standard OSA events, you can make it s*criptable* , or capable of responding to instructions written in a system-wide scripting language such as Object REXX. If necessary, your application can also send OSA events to itself or to other applications.

About the OSA Event Manager provides more information about the steps you need to take to support OSA events in your application.

The next section describes how OSA events can describe data and other items within an application or its documents. Subsequent sections describe the basic organization of OSA events and the data structures from which they are constructed.

OSA Events and OSA Event Objects

The Open Documents event shown in the figure found in the section About OSA Events, like the other three required events, specifies an

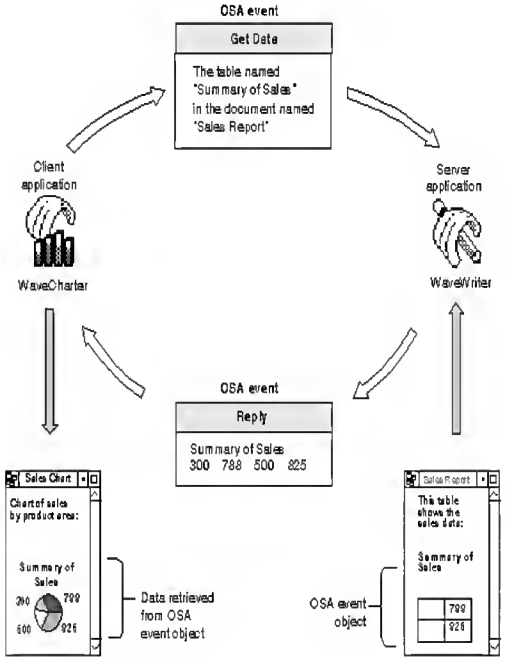
action and the applications or documents to which that action applies. The *OSA Event Registry: Standard Suite*s provides a vocabulary of actions for use by all applications. In addition to a vocabulary of actions, effective communication between applications requires a method of referring to windows, data (such as words or graphic elements), files, folders, volumes, and other discrete items on which actions can be performed. The OSA Event Manager includes routines that allow any application to construct or interpret "noun phrases" that describe the objects on which OSA events act.

Most of the OSA event definitions in the *OSA Event Registry: Standard Suite*s include definitions of *OSA event object classes* , which are simply names for objects that can be acted upon by each kind of OSA event. An *OSA event object* is any distinct item supported by an application that can be described within an OSA event. OSA event objects can be anything that an application can locate on the basis of such a description, including items that a user can differentiate and manipulate while using an application, such as words, paragraphs, shapes, windows, or style formats.

The definition for each OSA event object class in the *OSA Event Registry: Standard Suite*s determines only how that kind of OSA event object should be described within an OSA event, not how it should be represented internally by an individual application. You do not have to write your application in an object-oriented programming language to support OSA event objects. Instead, you need to organize your application so that it can interpret a request for specific OSA event objects, locate the objects, and perform the requested action on them.

The following figure shows a common OSA event, the Get Data event from the Core suite. In this example, the WaveCharter application is the client application; it requests data contained in a specific table in a WaveWriter document. To obtain the data it wants, the WaveCharter application must include a description of the data in the Get Data event it sends to WaveWriter. This description identifies the requested data as an OSA event object called a table. The table is named "Summary of Sales" and is located in a document named "Sales Report."

The WaveWriter application's Get Data handler extracts information about the request, locates the specified table, and returns a result. The OSA Event Manager provides a reply OSA event to which the WaveWriter application adds the requested information in the form requested by the Get Data event. The OSA Event Manager sends the reply event back to the WaveCharter application, which can use the requested data in whatever way is appropriate-in this case, displaying it as a pie chart.



OSA Event Attributes and Parameters

When an application creates and sends an OSA event, the OSA Event Manager uses arguments passed to OSA Event Manager routines to construct the data structures that make up the OSA event. An OSA event consists of attributes (which identify the OSA event and denote its task) and, often, parameters (which contain information to be used by the target application).

An *OSA event attribute* identifies the event class, event ID, target application, or some other characteristic of an OSA event. Taken together, the attributes of an OSA event denote the task to be performed on any data specified in the OSA event's parameters. A client application can use OSA Event Manager routines to add attributes to an OSA event. After receiving an OSA event, a server application can use OSA Event Manager routines to extract and examine its attributes.

An *OSA event parameter* contains data that the target application uses. Unlike OSA event attributes (which contain information that can be used by both the OSA Event Manager and the target application), OSA event parameters contain data used only by the target application. For example, the OSA Event Manager uses the event class and event ID attributes to identify the server application's handler for a specific OSA event, and the server application must have a handler to process the event identified by those attributes. By comparison, the list of documents

contained in a parameter to an Open Documents event is used only by the server application. As with attributes, a client application can use OSA Event Manager routines to add parameters to an OSA event, and a server application can use OSA Event Manager routines to extract and examine the parameters of an OSA event it has received.

Note that OSA event parameters are different from the parameters of OSA Event Manager functions. OSA event parameters are records used by the OSA Event Manager; function parameters are arguments you pass to the function or that the function returns to you. You can specify both OSA event parameters and OSA event attributes in parameters to OSA Event Manager functions. For example, the AEGetParamPtr function uses a buffer to return the data contained in an OSA event parameter. You can specify the OSA event parameter whose data you want in one of the parameters of the AEGetParamPtr function.

OSA Event Attributes

OSA events are identified by their event class and event ID attributes. The *event class* is the attribute that identifies a group of related OSA events. The event class appears in the message field of the event record for an OSA event. For example, the four required OSA events have the value **aevt** in the message fields of their event records. The value **aevt** can also be represented by the kCoreEventClass constant. Several event classes are shown here:

Event Class	Value	Description
kAECoreSuite	core	A core OSA event
kCoreEventClass	aevt	A required OSA event

The *event ID* is the attribute that identifies the particular OSA event within its event class. In conjunction with the event class, the event ID uniquely identifies the OSA event and communicates what action the OSA event should perform. (The event IDs appear in the where field of the event record for an OSA event. For example, the event ID of an Open Documents event has the value **odoc** (which can also be represented by the kAEOpenDocuments constant). The kCoreEventClass constant in combination with the kAEOpenDocuments constant identifies the Open Documents event to the OSA Event Manager.

Here are the event IDs for the four required OSA events:

Event ID	Value	Description
kAEOpenApplication	oapp	Perform tasks required when a user opens your application without opening or printing any documents
kAEOpenDocuments	odoc	Open documents
kAEPrintDocuments	pdoc	Print documents
kAEQuitApplication	quit	Quit the application

In addition to the event class and event ID attributes, every OSA event must include an attribute that specifies the target application's address. Remember that the target application is the one addressed to receive the OSA event. Your application can send an OSA event to itself or to another application.

Every OSA event must include event class, event ID, and target address attributes. Some OSA events can include other attributes; see Keyword-Specified Descriptor Records for a complete list.

OSA Event Parameters

As with attributes, there are various kinds of OSA event parameters. A *direct parameter* usually specifies the data to be acted upon by the target application. For example, the direct parameter of the Print Documents event contains a list of documents. Some OSA events also take *additional parameter*s , which the target application uses in addition to the data specified in the direct parameter. Thus, an OSA event for arithmetic operations might include additional parameters that specify operands in an equation.

The *OSA Event Registry: Standard Suite*s describes all parameters as either required or optional. A *required parameter* is one that must be present for the target application to carry out the task denoted by the OSA event. An *optional parameter* is a supplemental OSA event parameter that also can be used to specify data to the target application. Optional parameters need not be included in an OSA event; default

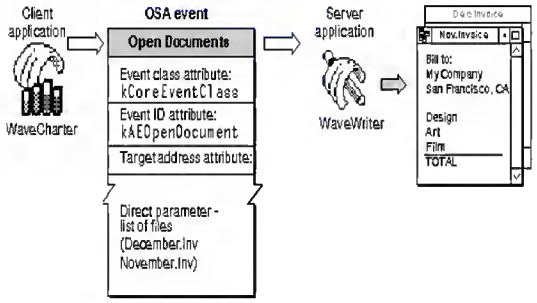
values for optional parameters are part of the event definition. The target application that handles the event must supply default values if the optional parameters are omitted.

Direct parameters are usually defined as required parameters in the *OSA Event Registry: Standard Suite*s ; additional parameters may be defined as either required or optional. However, the OSA Event Manager does not enforce the definitions of required and optional events. Instead, the source application specifies, when it sends the event, which OSA event parameters the target can treat as if they were optional.

For more information about optional parameters, see Specifying Optional Parameters for an OSA Event.

Interpreting OSA Event Attributes and Parameters

The following figure shows the major OSA event attributes and direct parameter for the Open Documents event introduced in the figure found in the section About OSA Events.



WM_SEMANTICEVENT is a PM message used to receive OSA events. When the WaveWriter application receives any OSA event it calls the AEProcessOSAEvent function to process the event. For an OSA event such as the Open Documents event shown in the previous figure, the AEProcessOSAEvent function uses the event class and event ID attributes to dispatch the event to the WaveWriter application's Open Documents handler. In response, the Open Documents handler opens the documents specified in the direct parameter.

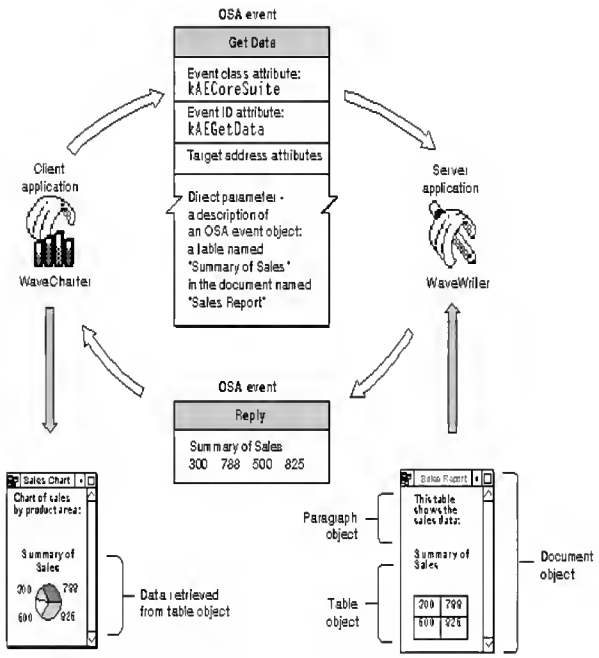
The definition of a given OSA event in the *OSA Event Registry: Standard Suite*s suggests how the source application can organize the data in the OSA event's parameters and how the target application interprets that data. The data in an OSA event parameter may use standard or private data types and may include a description of an OSA event object. Each OSA event handler provided by an application should be written with the format of the expected data in mind.

OSA events can use standard data types, such as strings of text, long integers, and Boolean values, for the corresponding data in OSA event parameters. For example, the Get Data event can contain an optional parameter specifying the form in which the requested data should be returned. This optional parameter always consists of a list of four-character codes denoting desired descriptor types in order of preference. OSA events can also use special data types defined by the OSA Event Manager.

OSA event parameters often contain descriptions of OSA event objects. These descriptions make use of a standard classification scheme summarized in Classification of OSA Event Objects.

For example, every Get Data event includes a required parameter that describes the OSA event object containing the data requested by the client application. Thus, one application can send a Get Data event to another application, requesting, for instance, one paragraph of a document, the first and last paragraphs of a document, all pictures in the document, all paragraphs containing the word "sales," or pages 10 through 12 of the document.

The following figure shows the OSA event attributes and direct parameter for the Get Data event introduced in the figure in section OSA Events and OSA Event Objects. The direct parameter for the Get Data event sent by the WaveCharter application describes the requested OSA event object as a table called "Summary of Sales" in the document "Sales Report." Both the table and the document are OSA event objects. The description of an OSA event object always includes a description of its location. In most cases, OSA event objects are located inside other OSA event objects.



To process the information in the Get Data event, the WaveWriter application calls the AEProcessOSAEvent function. The AEProcessOSAEvent function uses the event class and event ID attributes to dispatch the event to the WaveWriter application's handler for the Get Data OSA event. The WaveWriter application responds to the Get Data event by *resolving* the description of the OSA event object-that is, by using the AEResolve function, other OSA Event Manager routines, and its own application-defined functions to locate the table named "Summary of Sales." After locating the table, WaveWriter adds a copy of the table's data to the reply event, which the OSA Event Manager then sends to the WaveCharter application. The WaveCharter application then displays the data in its active window.

The next section describes the data structures the OSA Event Manager uses for OSA event attributes and parameters.

Data Structures within OSA Events

The OSA Event Manager constructs its own internal data structures to contain the information in an OSA event. Neither the sender nor the receiver of an OSA event should manipulate data directly after it has been added to an OSA event; each should rely on OSA Event Manager functions to do so.

This section describes the most important data structures used by the OSA Event Manager to construct OSA events. The first structure described is the descriptor record, a data structure of type AEDesc.

In some cases it is convenient for the OSA Event Manager to describe descriptor records by data types that indicate their contents; thus, it also defines data structures such as type AEAddressDesc, AEDescList, and AERecord, which are descriptor records used to hold addresses, lists of other descriptor records, and OSA event parameters, respectively. These and most of the other data structures described in this section are formally defined as data structures of type AEDesc; they differ only in the purposes for which they are used.

Descriptor Records

Descriptor records are the building blocks used by the OSA Event Manager to construct OSA event attributes and parameters. A *descriptor record* is a data structure of type AEDesc; it consists of a handle to data and a descriptor type that identifies the type of the data to which the handle refers.

```
typedef struct _AEDesc {
DescType        descriptorType;
Handle          dataHandle;
} AEDesc;
```

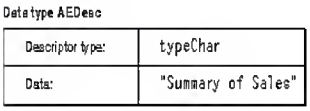
The fields in an AEDesc record should not be manipulated directly. The OSA Event Manager provides a set of APIs to manipulate these field.

The *descriptor type* is a structure of type DescType, which in turn is an unsigned long-that is, a four-character code. Constants are usually used in place of these four-character codes when referring to descriptor types. Descriptor types represent various data types. Here are some of the major descriptor type constants, their values, and the kinds of data they identify.

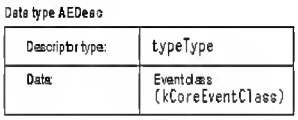
Descriptor Type	Value	Description of data
typeAEList	list	List of descriptor records
typeAERecord	reco	List of keyword-specified descriptor records
typeBoolean	bool	1-byte Boolean value
typeChar	TEXT	Unterminated string
typeEnumerated	enum	Enumerated data
typeKeyword	keyw	OSA event keyword
typeLongInteger	long	32-bit integer
typeMagnitude	magn	Unsigned 32-bit integer
typeNull	null	Nonexistent data (handle whose value is NULL)
typeOSAEvent	aevt	OSA event record
typeShortInteger	shor	16-bit integer
typeType	type	Four-character code

For a complete list of the basic descriptor types used by the OSA Event Manager, see the table in section Descriptor Records and Related Data Structures.

The following figure illustrates the logical arrangement of a descriptor record with a descriptor type of typeChar, which specifies that the data handle refers to an unterminated string (in this case, the text "Summary of Sales").



The following figure illustrates the logical arrangement of a descriptor record with a descriptor type of typeType, which specifies that the data handle refers to a four-character code (in this case, the constant kCoreEventClass, whose value is **aevt**). This descriptor record can be used in an OSA event attribute that identifies the event class for any OSA event in the Core suite.



Every OSA event includes an attribute specifying the address of the target application. A descriptor record that contains an application's address is called an *address descriptor record* .

```
typedef AEDesc AEAddressDesc;
```

The address in an address descriptor record can be specified as one of the following basic types:

Descriptor Type	Value	Description
typeHWND	HWND	Window handle
typePID	PID	Process ID

Like several of the other data structures defined by the OSA Event Manager for use in OSA event attributes and OSA event parameters, an address descriptor record is identical to a descriptor record of data type AEDesc; the only difference is that the data for an address descriptor record must always consist of an application's address.

Note: Programmers should not access the fields of an AEDesc structure directly. The internal format of the AEDesc structure will change in future releases of OSA; in particular, the current *dataHandl*e field will change from a handle (pointer to a pointer) to a simple pointer. Programmers should consider the AEDesc as an opaque, encapsulated structure and access it only through OSA Event Manager APIs to ensure that programs will be compatible with future releases.

- AEClearDesc
- AECreateDesc
- AEDisposeDesc
- AEDuplicateDesc
- AEGetDescData
- AESizeOfDescData

Keyword-Specified Descriptor Records

After the OSA Event Manager has assembled the necessary descriptor records as the attributes and parameters of an OSA event, your application cannot examine the contents of the OSA event directly. Instead, your application must use OSA Event Manager routines to request each attribute and parameter by keyword. *Keyword*s are arbitrary names used by the OSA Event Manager to keep track of various descriptor records. The AEKeyword data type is defined as a four-character code.

```
typedef unsigned long AEKeyword;
```

Constants are typically used for keywords. Here is a list of the keyword constants for OSA event attributes:

Attribute Keyword	Value	Description
keyAddressAttr	addr	Address of target or client application
keyEventClassAttr	evcl	Event class of OSA event
keyEventIDAttr	evid	Event ID of OSA event
keyEventSourceAttr	esrc	Nature of the source application
keyInteractLevelAttr	inte	Settings for allowing the OSA Event Manager to bring a server application to the foreground, if necessary, to interact with the user
keyMissedKeywordAttr	miss	Keyword for first required parameter remaining in an OSA event
keyOptionalKeywordAttr	optk	List of keywords for parameters of the OSA event that should be treated as optional by the target application
keyOriginalAddressAttr	from	Address of original source of OSA event if the event has been forwarded
keyReturnIDAttr	rtid	Return ID for reply OSA event
keyTimeoutAttr	timo	Length of time, in ticks, that the client will wait for a reply or a result from the server
keyTransactionIDAttr	tran	Transaction ID identifying a series of OSA events

Here is a list of the keyword constants for commonly used OSA event parameters:

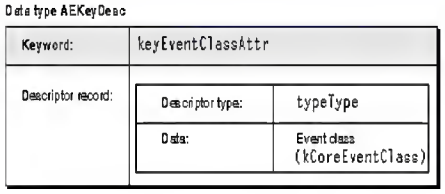
```
Parameter Keyword       Value   Description

keyDirectObject         ----    Direct parameter

keyErrorNumber          errn    Error number parameter

keyErrorString          errs    Error string parameter
```

The *OSA Event Registry: Standard Suites* defines additional keyword constants for OSA event parameters that can be used with specific OSA events.

The OSA Event Manager associates keywords with specific descriptor records by means of a *keyword-specified descriptor record* , a data structure of type AEKeyDesc that consists of a keyword and a descriptor record.

```
typedef struct _AEKeyDesc {
AEKeyword       descKey;
AEDesc          descContent;
} AEKeyDesc;
```

The following figure illustrates a keyword-specified descriptor record with the keyword keyEventClassAttr-the keyword that identifies an event class attribute. The figure shows the logical arrangement of the event class attribute for the Open Documents event shown in the figure in section Interpreting OSA Event Attributes and Parameters. The descriptor record in the following figure is identical to the one in the second figure in section Descriptor Records; its descriptor type is typeType, and the data to which its handle refers identifies the event class as kCoreEventClass.



Descriptor Lists

When extracting data from an OSA event, you use OSA Event Manager functions to copy data to a buffer specified by a pointer, or to return lists of descriptor records (called descriptor lists).

As previously noted, the descriptor record (of data type AEDesc) is the fundamental structure in OSA events, and it consists of a descriptor type and a handle to data. A *descriptor list* is a data structure of type AEDescList defined by the data type AEDesc-that is, a descriptor list is a descriptor record whose data handle refers to a list of other descriptor records (unless it is an empty list).

```
typedef AEDesc AEDescList;
```

Like several other OSA Event Manager data structures, a descriptor list is identical to a descriptor record of data type AEDesc; the only difference is that the data in a descriptor list must always consist of a list of other descriptor records.

The following figure illustrates the logical arrangement of the descriptor list that specifies the direct parameter of the Open Documents event shown in the figure in section Interpreting OSA Event Attributes and Parameters.

Data type AEDescList

Descriptor type:	typeAEList
Data:	**List of descriptor records:**

Descriptor type:	typeOS2FileName
Data:	File name (November.Inv)

Descriptor type:	typeOS2FileName
Data:	File name (December.Inv)

The descriptor list in the previous figure provides the data for a keyword-specified descriptor record. Keyword-specified descriptor records for OSA event parameters can in turn be combined in an *AE record* , which is a descriptor list of data type AERecord.

```
typedef AEDescList AERecord;
```

The handle for a descriptor list of data type AERecord refers to a list of keyword-specified descriptor records that can be used to construct OSA event parameters. The OSA Event Manager provides routines that allow your application to create AE records and extract data from them when creating or responding to OSA events.

An AE record has the descriptor type typeAERecord and can be coerced to several other descriptor types. An *OSA event record* is another special descriptor list of data type OSAEvent and descriptor type typeOSAEvent.

```
typedef AERecord OSAEvent;
```

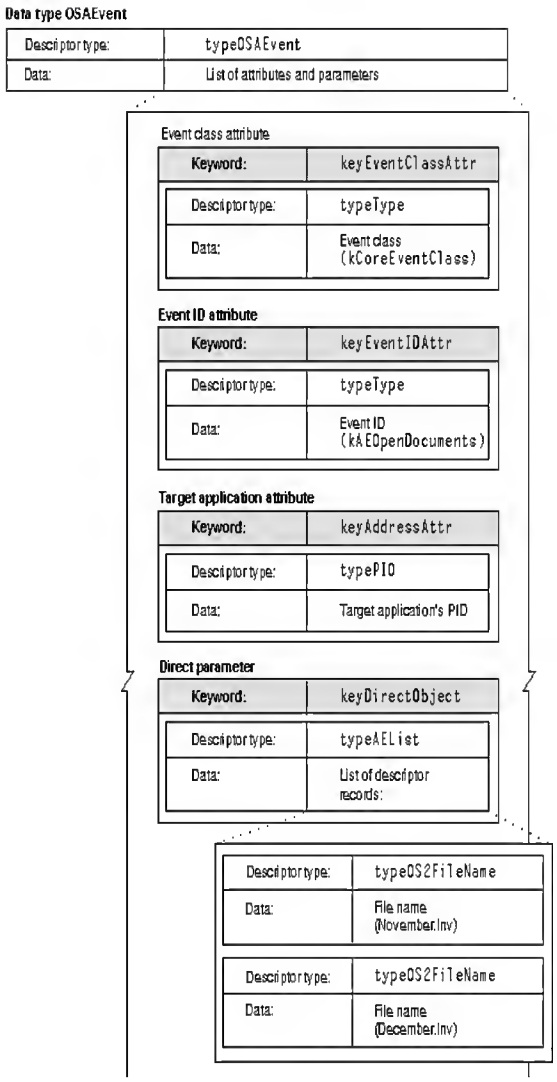
An OSA event record describes a full-fledged OSA event. Like the data for an AE record, the data for an OSA event record consists of a list of keyword-specified descriptor records. Unlike an AE record, the data for an OSA event record is divided into two parts, one for attributes and one for parameters. This division within the OSA event record allows the OSA Event Manager to distinguish between an OSA event's attributes and its parameters.

Descriptor lists, AE records, and OSA event records are all descriptor records whose handles refer to a nested list of other descriptor records. The data associated with each data type may be organized differently and is used by the OSA Event Manager for different purposes. In each case, however, the data is identified by a handle in a descriptor record. This means that you can pass an OSA event record to any OSA Event Manager function that expects an AE record. Similarly, you can pass OSA event records and AE records, as well as descriptor lists and descriptor records, to any OSA Event Manager functions that expect records of data type AEDesc.

When you use the AECreateOSAEvent function, the OSA Event Manager creates an OSA event record containing the attributes for an OSA event's event class, event ID, target address, return ID, and transaction ID. You then use OSA Event Manager functions such as AEPutParamDesc and AEPutAttributeDesc to add or modify attributes and to add any necessary parameters to the OSA event.

The following figure shows an example of a complete OSA event-a data structure of type OSAEvent containing a list of keyword-specified descriptor records that name the attributes and parameters of an Open Documents event. The figure includes the event class attribute shown in the figure in section Keyword-Specified Descriptor Records and the descriptor list shown in the previous figure, which forms the direct parameter-the keyword-specified descriptor record with the keyword keyDirectObject. The entire figure corresponds to the Open Documents event shown in the figure in section Interpreting OSA Event Attributes and Parameters.

The next two sections provide a quick overview of the steps your application must take to respond to and send OSA events.



Responding to OSA Events

A client application typically uses the OSA Event Manager to create and send an OSA event which requests a service or information. A server application responds by using the OSA Event Manager to process the OSA event, extract data from the attributes and parameters of the OSA event, and if necessary add requested data to the reply event returned by the OSA Event Manager to the client application. The server usually provides its own OSA event handler for performing the action requested by the client's OSA event.

As its first step in supporting OSA events, your application should support the required OSA events. Your application should also be able to respond to the standard OSA events that other applications are likely to send to it or that it can send to itself. This section provides a quick overview of the tasks your application must perform in responding to OSA events.

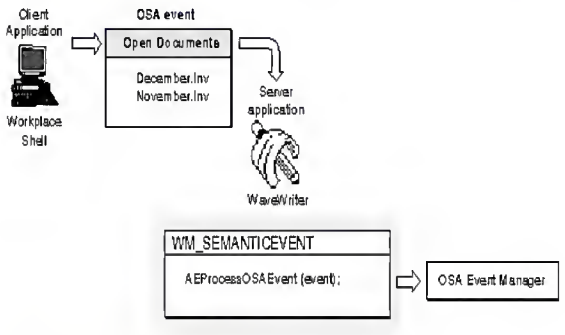
To respond to OSA events, your application must do the following:

- Create an object window to receive OSA events
- Use the AEInstallEventHandler function to install its OSA event handlers
- Provide OSA event handlers for the OSA events it supports
- Test for WM_SEMANTICEVENT in the object window's window procedure
- Use the AEProcessOSAEvent function to process OSA events

Accepting and Processing OSA Events

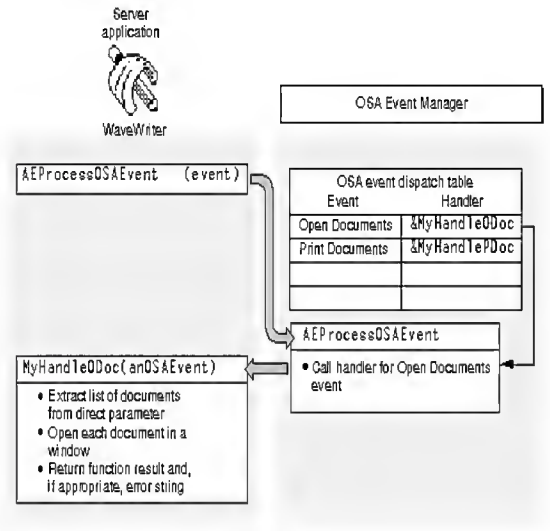
The semantic event record is identified by the *param1* field of the WM_SEMANTICEVENT message.

The AEProcessOSAEvent function is used to process an OSA event. The following figure shows how the WaveWriter application accepts and begins to process an OSA event sent by the Workplace Shell.



The AEProcessOSAEvent function first identifies the OSA event by examining the data in the event class and event ID attributes. The function then uses that data to call the OSA event handler that your application provides for that event. The OSA event handler extracts the pertinent data from the OSA event, performs the requested action, and returns a result. For example, if the event has an event class of kCoreEventClass and an event ID of kAEOpenDocuments, the AEProcessOSAEvent function calls your application's handler for the Open Documents event.

Before your application attempts to accept or process any OSA events, it must use the AEInstallEventHandler function to install OSA event handlers. This function installs handlers in an *OSA event dispatch table* for your application; the OSA Event Manager uses this table to map OSA events to handlers in your application. When your application calls the AEProcessOSAEvent function to process an OSA event, the OSA Event Manager checks the OSA event dispatch table and, if your application has installed a handler for that OSA event, calls that handler. The following figure shows how the flow of control passes from your application to the OSA Event Manager and back to your application.



About OSA Event Handlers

Your OSA event handlers must generally perform the following tasks:

- Extract the parameters and attributes from the OSA event
- Check that all the required parameters have been extracted
- Locate any OSA event objects specified by object specifier records in the OSA event parameters
- Perform the action requested by the OSA event
- Dispose of any copies of descriptor records that have been created
- Add information to the reply OSA event if requested

This section describes how your application's OSA event handlers can use the OSA Event Manager to accomplish some of these tasks. Responding to OSA Events provides detailed information about handling OSA events.

Extracting and Checking Data

You must use OSA Event Manager functions to extract the data from OSA events. You can also use OSA Event Manager functions to extract data from descriptor records, descriptor lists, and AE records. Most of these routines are available in two forms: they either return a copy of the data in a buffer or return a copy of the descriptor record for the data, including a copy of the data.

The following list shows the main functions you can use to access the data of an OSA event:

Function	Description
AEGetParamPtr	Uses a buffer to return a copy of the data contained in an OSA event parameter. Usually used to extract data of fixed length or known maximum length; for example, to extract the result code from the keyErrorNumber parameter of a reply OSA event.
AEGetParamDesc	Returns a copy of the descriptor record or descriptor list for an OSA event parameter. Usually used to extract data of variable length; for example, to extract the descriptor list for a list of file names specified in the direct parameter of the Open Documents event.
AEGetParamPtr	Uses a buffer to return a copy of the data contained in an OSA event attribute. Used to extract data of fixed length or known maximum length; for example, to determine the source of an OSA event by extracting the data from the keyEventSourceAttr attribute.
AEGetAttributeDesc	Returns a copy of the descriptor record for an attribute. Used to extract data of variable length; for example, to make a copy of a descriptor record containing the address of an application.
AECountItems	Returns the number of descriptor records in a descriptor list. Used, for example, to determine the number of file names for documents specified in the direct parameter of the Open Documents event.
AEGetNthPtr	Uses a buffer to return a copy of the data for a descriptor record contained in a descriptor list. Used to extract data of fixed length or known maximum length; for example, to extract the name and location of a document from the descriptor list specified in the direct parameter of the Open Documents event.
AEGetNthDesc	Returns a copy of a descriptor record from a descriptor list. Used to extract data of variable length; for example, to get the descriptor record containing an file name from the list specified in the direct parameter of the Open Documents event.

You can specify the descriptor type of the resulting data for these functions; if this type is different from the descriptor type of the attribute or parameter, the OSA Event Manager attempts to coerce it to the specified type. For example, if a parameter has a descriptor type of long integer, you can coerce the data to a string representation of the integer by using typeChar when extracting the data.

After extracting all known OSA event parameters, your handler should check that it retrieved all the parameters that the source application considered to be required. To do so, determine whether the keyMissedKeywordAttr attribute exists. If so, your handler has not retrieved all the required parameters, and it should return an error.

Although the *OSA Event Registry: Standard Suite*s defines OSA event parameters as either required or optional, the OSA Event Manager does not enforce the definitions of required and optional events. Instead, the source application specifies, when it sends the event, which OSA event parameters the target can treat as if they were optional. For more information about optional parameters, see Specifying Optional Parameters for an OSA Event.

If any of the OSA event parameters include object specifier records, your handler should use the AEResolve function, other OSA Event Manager routines, and your own application-defined functions to locate the corresponding OSA event objects. For more information about locating OSA event objects, see Working with Object Specifier Records.

Performing the Requested Action and Returning a Result

When your application responds to an OSA event, it should perform the standard action requested by that event. For example, your application should respond to the Open Documents event by opening the specified documents in titled windows just as if the user had selected each document and then chosen **Open** from the **File** menu.

Many OSA events can ask your application to return data. For instance, if your application is a spelling checker, the client application might

expect your application to return data in the form of a list of misspelled words. The second figure in section Data Structures within an Object Specifier Record shows a similar example: a Get Data event that asks the server application to locate a specific OSA event object and return the data associated with it.

If the client application requests a reply, the OSA Event Manager prepares a reply OSA event by passing a default reply OSA event to your handler. If the client application does not request a reply, the OSA Event Manager passes a *null descriptor record* -that is, a descriptor record of type typeNull whose data handle has the value NULL-to your handler instead of a default reply OSA event. The default reply OSA event has no parameters when it is passed to your handler, but your handler can add parameters to it. If your application is a spelling checker, for example, you can return a list of misspelled words in a parameter; however, your handler should check whether the reply OSA event exists before attempting to add any attributes or parameters to it. Any attempt to add an OSA event attribute or parameter to a null descriptor record generates an error.

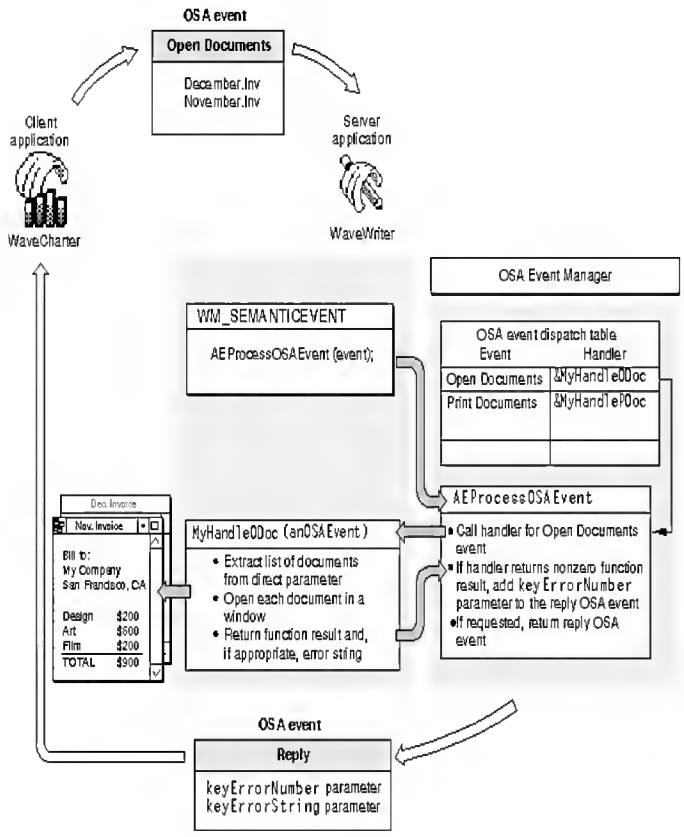
When you extract a descriptor record using the AEGetParamDesc, AEGetAttributeDesc, AEGetNthDesc, or AEGetKeyDesc function, the OSA Event Manager creates a copy of the descriptor record for you to use. When your handler is finished using a copy of a descriptor record, you should dispose of it-and thereby deallocate the memory used by its data-by calling the AEDisposeDesc function.

Note: Outputs from functions such as AEGetKeyPtr and other routines whose names end in -Ptr use a buffer rather than a descriptor record to return data. Because these functions do not require the use of AEDisposeDesc, it is preferable to use them.

Your OSA event handler should always set its function result either to noErr if it successfully handles the OSA event or to a nonzero result code if an error occurs. If your handler returns a nonzero result code, the OSA Event Manager adds a keyErrorNumber parameter to the reply OSA event (unless you have already added a keyErrorNumber parameter). This parameter contains the result code that your handler returns. The client should check whether the keyErrorNumber parameter exists to determine whether your handler performed the requested action. In addition to returning a result code, your handler can also return an error string in the keyErrorString parameter of the reply OSA event. The client can use this string in an error message to the user.

If the client application requested a reply, the OSA Event Manager returns the reply OSA event, which is identified by the event class kCoreEventClass and by the event ID kAEAnswer. When the client has finished using the reply OSA event, it should dispose of both the reply event and the original event-and thereby deallocate the memory they use-by calling the AEDisposeDesc function. The OSA Event Manager takes care of disposing both the OSA event and the reply OSA event after a server application's handler returns to AEProcessOSAEvent, but a server application is responsible for disposing of any OSA event data structures it creates while extracting data from the OSA event.

The following figure shows the entire process of responding to an OSA event.



When your handler returns a result code to the OSA Event Manager, you have finished your response to the client application's OSA event.

Creating and Sending OSA Events

Your application can use OSA events to request services or information from other applications, send information to other applications, or trigger actions within your application. For example, you can use the core OSA event Get Data to request specific data from another application's documents. Similarly, you can use other OSA events to request services-for example, asking a spell-checking application to check the text in a document created by your application. Consult the *OSA Event Registry: Standard Suites* for the format and function of the standard OSA events that you want your application to send.

To communicate with another application by sending an OSA event, your application must do the following:

- Create an OSA event record by calling the AECreateOSAEvent function
- Use OSA Event Manager functions to add parameters and any additional attributes to the OSA event
- call the AESend function to send the OSA event
- Dispose of any copies of descriptor records that you have created
- Handle the reply OSA event (if necessary)

The sections that follow describe how your application can use the OSA Event Manager to accomplish these tasks. Creating and Sending OSA Events provides detailed information about creating and sending OSA events.

To act as a server for your application, the target application must support OSA events. The server can be your own application or another application running on your computer.

Creating an OSA Event Record

Use the AECreateOSAEvent function to create an OSA event record. Using the arguments you pass to the AECreateOSAEvent function, the OSA Event Manager constructs the data structures describing the event class, the event ID, and the target address attributes of an OSA event. The event class and event ID, of course, identify the particular event you wish to send. The target address identifies the intended recipient of the OSA event.

You can specify two other attributes with the AECreateOSAEvent function: the reply ID and the transaction ID. For the reply ID attribute, you usually specify the kAutoGenerateReturnID constant to the AECreateOSAEvent function. This constant ensures that the OSA Event Manager generates a unique return ID for the reply OSA event returned from the server. For the transaction ID attribute, you usually specify the kAnyTransactionID constant, which indicates that this OSA event is not one of a series of interdependent OSA events.

Adding OSA Event Attributes and Parameters

The OSA event record created with the AECreateOSAEvent function serves as a foundation for the OSA event you want to send. Descriptor records and descriptor lists are the building blocks from which the complete OSA event record is constructed. To create descriptor records and descriptor lists and add items to a descriptor list, use the following functions:

Function	Description
AECreateDesc	Takes a descriptor type and a pointer to data and converts them into a descriptor record
AECreateList	Creates an empty descriptor list or AE record.
AEPutPtr	Takes a descriptor type and a pointer to data and adds the data to a descriptor list as a descriptor record; used, for example, to add to a descriptor list a number used as the parameter of an OSA event requesting a calculation.
AEPutDesc	Adds a descriptor record to a descriptor list; used, for example, to add to a descriptor list a file name used as the direct parameter of an OSA event requesting file manipulation.

To add the remaining attributes and parameters necessary for your OSA event to the OSA event record, you can use these additional OSA Event Manager functions:

Function	Description
AEPutParamPtr	Takes a keyword, descriptor type, and pointer to data and adds the data to an OSA event record as a parameter with the specified keyword (replacing any existing parameter with the same keyword); used, for example, to put numbers into the parameters of an OSA event that asks the server to perform a calculation.

AEPutParamDesc	Takes a keyword and a descriptor record and adds the descriptor record to an OSA event record as a parameter with the specified keyword (replacing any existing parameter with the same keyword); used, for example, to place a descriptor list containing file names into the direct parameter of an OSA event that requests a server to manipulate files.
AEPutAttributePtr	Takes a keyword, descriptor type, and pointer to data and adds the descriptor record to an OSA event record as an attribute with the specified keyword (replacing any existing attribute with the same keyword); used, for example, to change the transaction ID of an OSA event record that is waiting to be sent.
AEPutAttributeDesc	Takes a keyword and a descriptor record and adds the descriptor record to an OSA event record as an attribute with the specified keyword (replacing any existing attribute with the same keyword); used, for example, to replace the descriptor record used for the target address attribute in an OSA event record waiting to be sent.

OSA event parameters for core events and functional-area events can include descriptions of OSA event objects in special descriptor records called object specifier records. For an overview of object specifier records, see Working with Object Specifier Records.

Sending an OSA Event and Handling the Reply

After you add all the attributes and parameters required for the OSA event, use the AESend function to send the OSA event. The OSA Event Manager uses the Presentation Manager to transmit the OSA event to the server application.

The AESend function requires that you specify whether your application should wait for a reply from the server. If you specify that you want a reply, the OSA Event Manager prepares a reply OSA event for your application by passing a default reply OSA event to the server. The OSA Event Manager returns any nonzero result code from the server's handler in the keyErrorNumber parameter of the reply OSA event. The server can return an error string in the keyErrorString parameter of the reply OSA event. The server can also use the reply OSA event to return any data you requested-for example, the results of a numeric calculation or a list of misspelled words.

You specify how your application should wait for a reply by using one of these flags in the *sendMode* parameter of the AESend function:

Flag	**Description**
kAENoReply	Your application does not want a reply OSA event.
kAEQueueReply	Your application wants a reply OSA event; the reply appears in your event queue as soon as the server has the opportunity to process and respond to your OSA event.
kAEWaitReply	Your application wants a reply OSA event and is willing to wait for it. When using the kAEWaitReply flag, your application must create a separate thread for the AESend function call. This will allow the main thread to service other incoming events sent to the object window's PM message queue.

Important: If the kAEWaitReply flag is specified, the application must wait for the reply in a separate thread. This allows the main thread to service the PM message queue.

If the kAENoReply flag is specified, the reply OSA event prepared by the OSA Event Manager for the server application consists of a null descriptor record.

After you send an OSA event, your application is responsible for disposing of the OSA event record-and thereby deallocating the memory its data uses-by calling the AEDisposeDesc function. If you create one descriptor record and add it to another, the OSA Event Manager adds a copy of the newly created one to the existing one and also makes a copy of the associated data. For example, you might use the AECreateDesc function to create a descriptor record that you wish to add to an OSA event. When you use the AEPutParamDesc function, it adds a copy of your newly created descriptor record, including its data, as a parameter to an existing OSA event. When you no longer need the original descriptor record, you should call AEDisposeDesc to dispose of it.

Your application should dispose of all the descriptor records that are created for the purposes of adding parameters and attributes to an OSA event. You normally dispose of your OSA event and its reply after you receive a result from the AESend function. You should dispose of these even if AESend returns an error result.

If you specify the kAEWaitReply flag, the reply OSA event is returned in a parameter you pass to the AESend function. If you specify the kAEQueueReply flag to the AESend function, the reply OSA event is returned in the PM message queue. In this case, the reply is identified by the event class kCoreEventClass and the event ID kAEAnswer. Your application processes reply events in its event queue in the same way that server applications process OSA events.

Your application should check for the existence of the keyErrorNumber parameter of the reply OSA event to ensure that the server performed the requested action. The server can also return, in the keyErrorString parameter, any error messages you need to display to the user.

Whenever a server application provides an error string, it should also provide an error number. However, you cannot count on all server applications to do so. The absence of the keyErrorNumber parameter does not necessarily mean that there will not an error string provided in

the keyErrorString parameter. A client application should therefore check for both the keyErrorNumber and keyErrorString parameters before assuming that no error has occurred. If a string has been provided without an error number, an error has occurred.

After extracting the information it needs from the reply event, your handler should dispose of the reply by calling the AEDisposeDesc function. Similarly, when your handler no longer needs descriptor records it has extracted from the reply, it should call AEDisposeDesc to dispose of them.

The next section provides an overview of the way a source application identifies OSA event objects supported by a target application. If you are starting by supporting only the Required suite, you can skip the next section and go directly to About the OSA Event Manager.

Working with Object Specifier Records

Most of the standard OSA events allow the source application to refer, in an OSA event parameter, to OSA event objects within the target application or its documents. The OSA Event Manager allows applications to construct and interpret such references by means of a standard classification system for OSA event objects. This system, described in detail in the *OSA Event Registry: Standard Suite*s , is summarized in Classification of OSA Event Objects. A description in an OSA event parameter that uses this classification system takes the form of an object specifier record.

An *object specifier record* is a descriptor record of descriptor type typeObjectSpecifier that describes the location of one or more OSA event objects: for example, the table "Summary of Sales" in the document "Sales Report," or the third row in that table, or the last row of the column "Totals." With the aid of application-defined functions, the OSA Event Manager can conduct a step-by-step search according to such instructions in an object specifier record, locating first the document, then the table, then other objects, and so on until the requested object has been identified. Object specifier records can specify many combinations of identifying characteristics that cannot be specified using one of the simple data types.

This section introduces object specifier records and the organization of their data. You need to read this section (a) if you plan to support the Core suite or any of the standard functional-area suites and (b) if you want to make your application scriptable-that is, capable of responding to scripts written in a scripting language.

Important: An object specifier record identifies one or more OSA event objects among many; it contains a description of each object, not the object itself. An OSA event object described by an object specifier record exists only in the server application's document or in the server application itself.

A client application cannot retrieve an OSA event object from a server application unless the server application can accurately locate it. Thus, to locate characters of a specific color, a server application must be able to identify a single character's color; to locate a character in a cell, a server application must be able to locate both the table and the cell.

A client application can create object specifier records for use as OSA event parameters. Scripting components can also create object specifier records as OSA event parameters for the OSA events they generate in the course of executing a script. A server application that receives an OSA event containing an object specifier record should resolve the object specifier record-that is, locate the requested OSA event objects.

To respond to core and functional-area OSA events received by your application, you must first define a hierarchy of OSA event objects for your application that you want other applications or scripting languages to be able to describe. The OSA event objects for your application should be based as closely as possible on the standard object classes described by the *OSA Event Registry: Standard Suite*s . After you have decided which of the standard OSA event objects make sense for your application, you can write functions that locate objects on the basis of information in an object specifier record. If you want your application to send specific OSA events to other applications, you must also write functions that can create object specifier records and add them to OSA events. Your application does not need to create object specifier records in order to be scriptable. However, to write functions that can help the OSA Event Manager resolve object specifier records, you need to know how they are constructed.

The section Finding OSA Event Objects provides an overview of the way the OSA Event Manager works with your application-defined functions to locate the OSA event objects described in an object specifier record. Resolving and Creating Object Specifier Records describes in detail how to support object specifier records as a server or client application.

Data Structures within an Object Specifier Record

The organization of the data for an object specifier record is nearly identical to that of the data for an AE record. An object specifier record is a structure of data type AEDesc whose data handle usually refers to four keyword-specified descriptor records describing one or more OSA event objects. An AE record is a structure of data type AERecord whose data handle refers to one or more OSA event parameters.

The four keyword-specified descriptor records for an object specifier record provide information about the requested OSA event object or objects.

Keyword	**Description**

keyAEDesiredClass	Four-character code indicating the object class ID
keyAEContainer	A description of the container for the requested object, usually in the form of another object specifier record
keyAEKeyForm	Four-character code for the key form, which indicates how to interpret the key data
keyAEKeyData	Key data, used to distinguish the desired OSA event object from other objects of the same object class in the same container

For example, the data for an object specifier record identifying a table named "Summary of Sales" in a document named "Sales Report" consists of four keyword-specified descriptor records that provide the following information:

- The object class ID for a table
- Another object specifier record identifying the document "Sales Report" as the container for the table
- A key form constant indicating that the key data contains a name
- Key data that consists of the string "Summary of Sales"

The *object class ID* specifies the OSA event object class to which the object belongs. An OSA event object class is a category for OSA event objects that share specific characteristics (see OSA Events and OSA Event Objects) The characteristics of each object class are listed in the *OSA Event Registry: Standard Suite*s . For example, the Core suite defines object classes for documents, paragraphs, words. and windows. The first keyword-specified descriptor record in an object specifier record uses a four-character code or a constant to specify the object class ID. The object class for words, for example, can be identified by either the object class ID **rowc** or the constant cWord.

Note: The object class ID identifies the object class of an OSA event object described in an object specifier record, whereas the event class and event ID identify an OSA event.

The *container* for an OSA event object is usually another OSA event object. For example, the container for a document might be a window, and the container for characters, delimited items, or a word might be another word, a paragraph, or a document. The container is identified by the second keyword-specified descriptor record in an object specifier record; usually this is another object specifier record. The container can also be specified by a null descriptor record, which indicates a default container or a container already known to the OSA Event Manager.

The descriptor record in an object specifier record that identifies an OSA event object's container can in turn use another object specifier record to identify the container's container, and so on until the OSA event object is fully specified. For example, an object specifier record identifying a paragraph might specify the paragraph's container with another object specifier record that identifies a page. That object specifier record might in turn specify the page's container with another object specifier record identifying a document. The ability to nest one object specifier record within another in this way makes it possible to identify elements such as "the first row in the table named 'Summary of Sales' in the document named 'Sales Report.'"

The *key form* and *key data* distinguish the desired OSA event object from other OSA event objects of the same object class. The key form describes the form the key data takes. The third keyword-specified descriptor record in an object specifier record usually specifies the key form with one of seven standard constants shown in the following table.

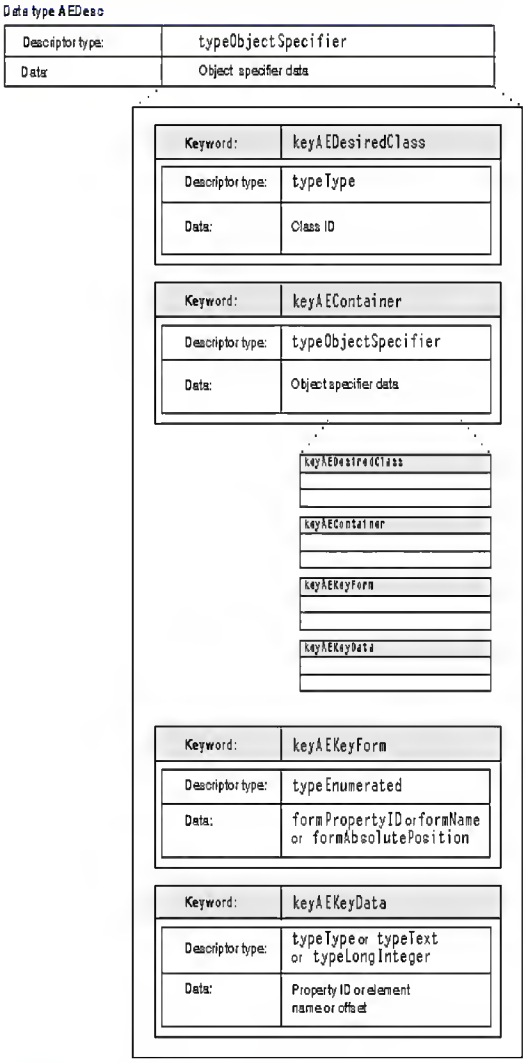
Key Form	Value	Corresponding Key Data
formAbsolutePosition	indx	An integer or other constant indicating the position of one or more elements in relation to the beginning or end of their container
formName	name	Element's name
formPropertyID	prop	Property ID for an element's property
formRelativePosition	rele	A constant that specifies the element just before or after the container
formRange	rang	Descriptor records that specify a group of elements between two other elements
formTest	test	Descriptor records that specify a test
formUniqueID	ID	A value that uniquely identifies an object within its container or across an application

A key form of formPropertyID indicates key data that specifies a property. A *property* of an OSA event object is a specific characteristic of that

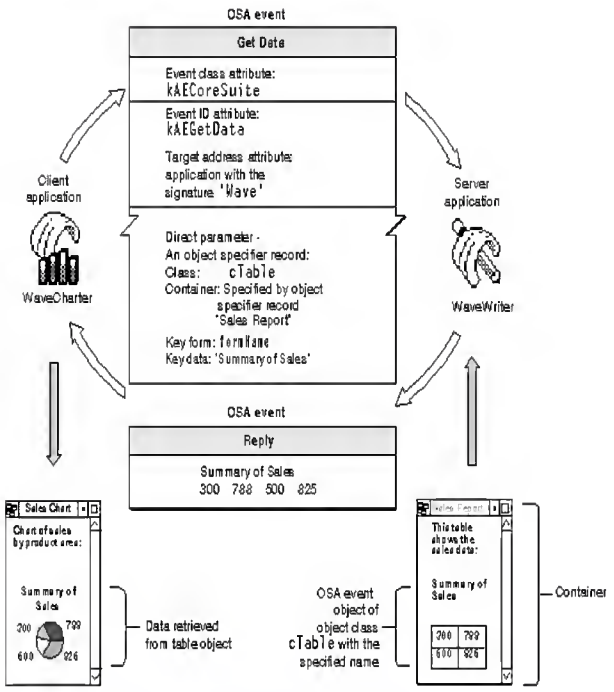
object that can be identified by a constant. The properties associated with the object class for documents include the name of the document and a flag indicating whether the document has been modified since the last save. The properties associated with the object class for words include color, font, point size, and style.

The following figure shows the structure of a typical object specifier record: four keyword-specified descriptor records that specify the class ID, the container, the key form, and the key data. These four keyword-specified descriptor records are the data for a descriptor record (AEDesc) of descriptor type typeObjectSpecifier. Note the similarities between the object specifier record shown in the following figure and the OSA event record shown in the second figure in section Descriptor Lists. Like an OSA event record or an AE record, an object specifier record consists of a list of keyword-specified descriptor records.

The following figure shows the structure of a simple object specifier record that specifies the key form formPropertyID, formName, or formAbsolutePosition. For detailed information about the structure of object specifier records that specify the other key forms, see the section Resolving and Creating Object Specifier Records.



The following figure shows the object specifier record for the Get Data event previously illustrated in the second figure in section Interpreting OSA Event Attributes and Parameters. The object class ID tells the WaveWriter application that the requested data is an element of class cTable. The container for the table is the document "Sales Report." The key form is formName, which tells the server application that the key data identifies the OSA event object by name. The key data is the name of the table.



To add an object specifier record to an OSA event as one of its parameters, your application must first create the object specifier record. The section Creating Object Specifier Records describes the OSA Event Manager routines for creating object specifier records.

To respond to OSA events that include object specifier records, your application should use the standard classification system for OSA event objects and provide functions that can locate those objects within your application or its documents. The next section summarizes the classification of OSA event objects as defined in the *OSA Event Registry: Standard Suites* .

Classification of OSA Event Objects

To create or resolve object specifier records, your application should use the classification of OSA event objects defined by the *OSA Event Registry: Standard Suites* . This section summarizes the concepts that underlie that classification system. You should have a copy of the *OSA Event Registry: Standard Suites* available for reference purposes while you read this section.

You do not need to write your application in an object-oriented programming language in order to support OSA event objects in your application. However, you must understand the classification system described in this section in order to classify OSA event objects in your application and to write routines that can locate them on the basis of information contained in object specifier records.

Object Classes

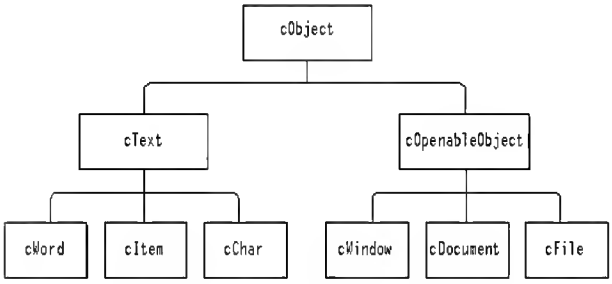
Except for the concept of inheritance, OSA event objects are different from the objects used in object-oriented programming languages. OSA event objects are distinct items in a server application or any of its documents that can be specified by an object specifier record in an OSA event sent by a client application. OSA event objects are often, but not always, items that a user can differentiate and manipulate within an application, such as words, paragraphs, shapes, windows, or style formats. Every OSA event object can be classified according to its object class, which defines both its characteristics and its behavior. The object classes listed in the *OSA Event Registry: Standard Suites* provide a method of describing OSA event objects that all applications can understand. Object classes permit more flexibility than simple descriptor types; for example, a word can be defined as a simple string, or it can be defined as an OSA event object with specific characteristics such as font or style.

Note: The definition of an object class only specifies conventions that determine how applications should handle OSA event objects that belong to that class. Your application must make sure that it uses the conventions correctly; they are not enforced by the OSA Event Manager.

Each object class is identified by a four-character object class ID, which can also be represented by a constant. Constants for object classes always begin with the letter c.

The definition of an object class specifies its *superclass* , which is the object class from which a *subclass* (the class being defined) inherits some of its characteristics. Characteristics can also be inherited from special object classes, called *abstract superclasses* , that are used only

in definitions of object classes and do not refer to real OSA event objects. The pattern of inheritance among object classes is called the *object class inheritance hierarchy* . The following figure shows a portion of this hierarchy. The abstract superclass cObject is at the top of the hierarchy and is therefore the only object class that has no superclass. At the next level are cText, which is a regular object class, and cOpenableObject, which is an abstract superclass. Both are subclasses of cObject and superclasses for their own subclasses. The object classes cWord, cItem, and cChar are all subclasses of cText. Similarly, cWindow, cDocument, and cFile are subclasses of cOpenableObject. Every object class inherits all the characteristics of its superclass and can also add characteristics of its own.



The following table shows some of the object classes defined for the Core suite.

Class	Class ID	Description
cChar	cha	Text characters
cDocument	docu	OS/2 documents
cFile	cfil	OS/2 files
cSelection	csel	User or application selections
cText	ctxt	Series of characters
cWindow	cwin	Standard OS/2 windows

The following table shows some of the object classes defined for the Text suite.

Class	Class ID	Description
cChar	cha	Text characters
cLine	clin	Lines of text
cParagraph	cpar	Paragraphs
cText	ctxt	Series of characters
cTextFlow	cflo	Text flows
cWord	cwor	Words

As you can see, some object classes, such as cChar and cText, are defined in more than one suite. For example, the definition of the cText object class in the Text suite is an *extension* of the cText object class defined in the Core suite; it duplicates all the characteristics of the Core suite object class and adds some of its own. Like a word in a dictionary, one object class ID can have several related definitions. You can choose to support the definition that best suits your application; or, if necessary, you can create extensions of your own. The extension of an object class is different from inheritance between object classes. An extension of a standard object class provides additional ways of describing an OSA event object of that class, whereas the object class inheritance hierarchy determines the pattern of characteristics shared by different object classes.

The definition of an object class always specifies a default descriptor type. Suppose, for example, that a client application sends a Get Data, Cut, or Copy event that specifies an OSA event object but does not specify a descriptor type for the returned data. In this case, the server application returns a descriptor record of the default descriptor type for the object class of the specified OSA event object. For example, the default descriptor type for OSA event objects of class cWord is typeOS2Text, a descriptor type that specifies an undelimited string of characters and the code page. The client application can also request that the data be returned in a descriptor record of some other data type.

The definition of an object class includes three lists of characteristics: properties, element classes, and OSA events that support the object class. (The next section describes properties and element classes.) Any or all of these characteristics may be inherited from a superclass. An OSA event is listed for an object class if its parameters can specify objects of that class. The definition for cWindow, for example, lists 12 OSA events, including the Open, Close, and Move events, whose parameters can include object specifier records that specify windows. The cWindow class inherits all of these OSA events from its abstract superclass, cOpenableObject.

The *OSA Event Registry: Standard Suite*s also defines *primitive object cla*sses , which describe OSA event objects that contain a single value. For example, the cBoolean, and cLongInteger object classes are all primitive object classes. The object class ID for a primitive object class is the same as the four-character value of its descriptor type. Primitive object classes contain no properties; they contain only the value of the data.

Properties and Elements

The properties listed for an object class can be used to identify characteristics of OSA event objects that belong to that class. Each property is identified by a four-character property ID, which can also be represented by a constant. Constants for properties always begin with the letter p.

The following table shows the constants and property IDs for some properties.

Property	Property ID	Description
pBounds	pbnd	Coordinates of a window
pClass	pcls	Class ID of an OSA event object
pColor	colr	Text color
pFillColor	flcl	Fill color
pFont	font	Font
pIsModal	pmod	Indicates whether a window is modal
pName	pnam	Name of an OSA event object
pTextPointSize	ptps	Point size
pTextStyle	txst	Text style
pVisible	pvis	Indicates whether a window is visible

The property of an OSA event object is itself defined as a single OSA event object whose container is the object to which the property belongs. For example, the pFont property of a word is defined by the name of a font, such as Helvetica the string that identifies the font is an OSA event object of class cText.

The constant cProperty specifies the object class for any object specifier record that identifies a property.

```
#define cProperty      0x706F7270      /* 'prop' */
```

An object specifier record for a property specifies cProperty as the object class ID, the OSA event object to which the property belongs as the container, formPropertyID as the key form, and a constant such as pFont as the key data.

The *element*s of a specific OSA event object are the other OSA event objects it contains, excluding those that define its properties. An object specifier record for an element specifies the OSA event object in which the element is located as the container and can specify any key form except formPropertyID. Each object class definition in the *OSA Event Registry: Standard Suite*s includes a list of *element cla*sses , which are the object classes of the elements that an OSA event object can contain.

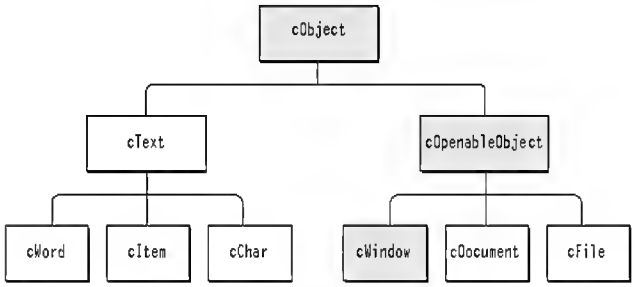
An OSA event object contains exactly one of each of its properties, whereas it can contain no elements or many elements of the same element class. In general, a property of an object describes something about that object; a property can be examined or changed but never deleted. An element can be one or more discrete objects contained in another object and can usually be deleted.

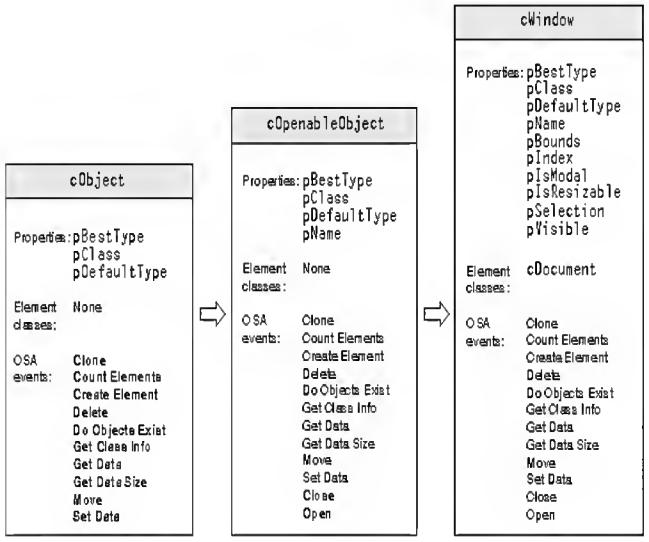
For example, because a paragraph can contain one or more words, one of the element classes listed for the object class cParagraph is the object class cWord. Individual words can be deleted from a paragraph. However, even though a word in a paragraph can be in a different font from the words around it, a paragraph can have only one pFont property. This property is defined as the font of the first character in the paragraph and consists of the name of a font. The paragraph's pFont property can be changed but not removed.

The properties and element classes listed for each object class definition in the *OSA Event Registry: Standard Suite*s can be inherited from a superclass, or they can originate with a subclass. The following figure illustrates the object class inheritance hierarchy for the object class cWindow in the Core suite. Boldface terms in the figure represent those properties, element classes, or OSA events that are not inherited. The object class cWindow includes all the properties and OSA events of its superclass, cOpenableObject, which in turn includes all the properties

and OSA events of its superclass, cObject. The object class cWindow also includes 11 properties and one element class that originate with cWindow and are not inherited.

The pClass property-the property that specifies the four-character class ID-originates with cObject. Because the definitions of all object classes are ultimately derived from cObject, pClass is inherited by all object classes. The definition for cObject also lists ten OSA events, which include common events such as Get Data, Move, and Delete Element. Because cObject is at the top of the object class inheritance hierarchy, these ten OSA events can use object specifier records that describe OSA event objects of any object class as a direct parameter. Like all abstract superclasses, cObject does not correspond to a real OSA event object, so its definition does not list any element classes. Unlike any other object class, cObject is at the top of the object class inheritance hierarchy and therefore does not have a superclass.



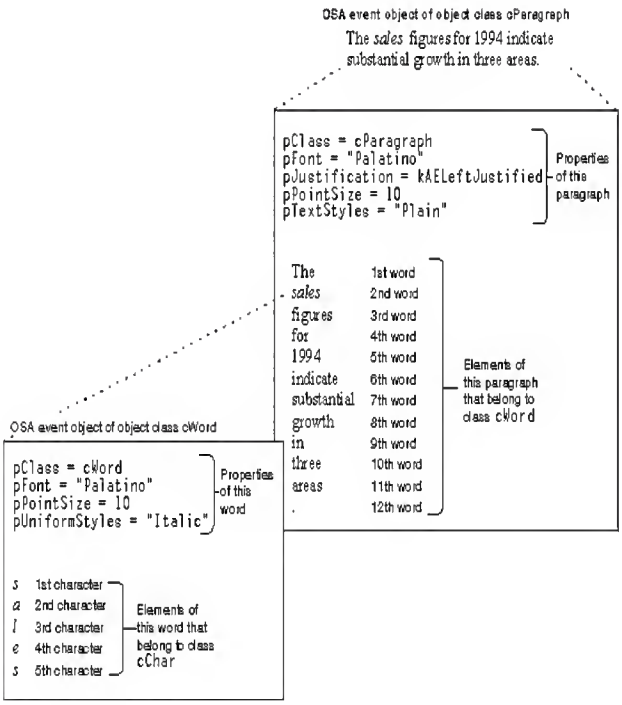


The chain of containers that determine the location of one or more OSA event objects is called the *container hierarchy* . The container hierarchy, which specifies the location of real OSA event objects, is different from the object class inheritance hierarchy, which is an abstract concept that determines which properties, element classes, and OSA events an object class inherits from its superclass. For example, the container hierarchy for an OSA event object of class cWord can vary from one word to another, because various combinations of other OSA event objects, such as a document, a paragraph, a delimited string, or another word, can contain a word.

Applications that support OSA event objects must be able to identify the order of several elements of the same class that are contained within another OSA event object. For example, each word in a paragraph should have an identifiable order, such as the 5th word or the 12th word. This allows other applications to identify OSA event objects by describing their absolute position within a container.

The following figure shows an OSA event object of object class cWord-the word "Sales"-contained in another OSA event object of object class cParagraph. (Both these object classes are defined in the Text suite.) The figure shows only a portion of the container hierarchy for the word, since a complete description of the word would also include the containers that specify the location of the paragraph.

Your application must take account of the definitions in the *OSA Event Registry: Standard Suite*s for any object classes you want to support. For example, the definition for the object class cText lists paragraphs, lines, words, and characters as OSA event objects that can be contained in OSA event objects of class cText. To support OSA events that refer to elements of object class cText, your application should associate the cText object class with paragraphs, lines, words, and characters in its documents. The list of properties defined for class cText includes the properties pColor, pFont, pPointSize, pCodePage, and pTextStyles. If you want to support OSA events that distinguish a boldface 12-point word of object class cText from an italic 14-point word, for example, your application must associate the point size and style of text in its documents with the properties pPointSize and pTextStyles defined for class cText.



Finding OSA Event Objects

Most of the OSA events in the Core suite and the functional-area suites defined in the *OSA Event Registry: Standard Suite*s can include parameters that consist of object specifier records. Your application's handlers for these events can use the AEResolve function to resolve object specifier records: that is, to locate the OSA event objects they describe.

The AEResolve function parses an object specifier record and performs related tasks that are the same for all object specifier records. When necessary, the AEResolve function calls application-defined functions to perform tasks that are unique to the application, such as locating a specific OSA event object in the application's data structures.

Your application can provide two kinds of application-defined functions for use by AEResolve. *Object accessor function*s locate OSA event objects. Every application that supports simple object specifier records must provide one or more object accessor functions. *Object callback function*s perform other tasks that only an application can perform, such as counting, comparing, or marking OSA event objects.

Each time AEResolve calls one of your application's object accessor functions successfully, the object accessor function returns a special descriptor record, called a *token* , that identifies either an element in a specified container or a property of a specified OSA event object. The token can be of any descriptor type, including descriptor types you define yourself.

You install object accessor functions by using the AEInstallObjectAccessor function. Much like the AEInstallEventHandler function, AEInstallObjectAccessor uses an *object accessor dispatch table* to map requests for OSA event objects to the appropriate object accessor functions in your application. These requests refer to objects of a specified object class in containers identified by a token of a specified descriptor type.

Responding to an OSA event that includes an object specifier record typically involves these steps:

1. After determining that the event is an OSA event, your application calls AEProcessOSAEvent.
2. The AEProcessOSAEvent function uses the OSA event dispatch table to dispatch the event to the your application's handler for that event.
3. The OSA event handler extracts the OSA event parameters, and passes the object specifier records they contain to AEResolve.
4. The AEResolve function uses the object accessor dispatch table to call one or more object accessor functions, one at a time, that can identify the nested OSA event objects described by each object specifier record. Each object accessor function returns a token for the object it finds, which in turn helps to determine which object accessor function the AEResolve function will use to locate the next OSA event object.
5. The AEResolve function returns the final token for the requested object to the application's handler.

The resolution of an object specifier record always begins with the outermost container it specifies. For example, to locate a table named "Summary of Sales" in the document named "Sales Report," the AEResolve function first calls an object accessor function that can locate the

document, then uses the token returned by that function to identify an object accessor function that can locate the table. It then returns the token for the table to the OSA event handler that called AEResolve.

Resolving and Creating Object Specifier Records describes in detail how to resolve object specifier records and how to write and install object accessor and object callback functions.

About the OSA Event Manager

You can use the OSA Event Manager to do the following:

- Respond to OSA events as a server application
- Create and send OSA events as a client application
- Resolve and create object specifier records
- Support OSA event recording

This section briefly summarizes the steps involved in providing each kind of support and tells where to find the relevant information in this book.

Supporting OSA Events as a Server Application

You do not need to implement all OSA events at once. You can begin by supporting just the required events. The beginning of the section Handling OSA Events describes how to provide this minimal level of support.

It is relatively easy to respond to the required events. If, however, your application cannot respond to any other OSA events, other applications will not be able to request services that involve locating specific OSA event objects within your application or its documents, and users will not be able to control your application by executing scripts. To respond to OSA events it is likely to receive from other applications or from scripting components, your application must be able to respond to the appropriate core and functional-area OSA events.

Once you have provided the basic level of support for the required suite, you should:

- Decide which other OSA event suites you want to support
- Define the hierarchy of OSA event objects within your application that you want scripting components or other applications to be able to identify-that is, which OSA event objects can be contained by other OSA event objects in your application
- Write handlers for the OSA events you support, and install corresponding entries in your application's OSA event dispatch table

To decide which OSA event suites you want to support and how to define the hierarchy of OSA event objects in your application, consult the *OSA Event Registry: Standard Suites* and evaluate which OSA events and OSA event object classes make sense for your application. If necessary, you can extend the definitions of the standard OSA events and OSA events objects to cover special requirements of your application. It is better to extend the standard definitions rather than to define your own custom OSA events, because only those applications that choose to support your custom OSA events explicitly will be able to make use of them.

Responding to OSA Events describes how to write OSA event handlers and related routines. Resolving and Creating Object Specifier Records describes how to resolve object specifiers in an OSA event that describes OSA event objects in your application or its documents.

If your application can respond to OSA events, you can make it scriptable simply by adding an **aete** resource. Scripting components use your application's **aete** resource to obtain information about the OSA events and corresponding human-language terminology that your application supports. OSA Event Terminology Resources describes how to optimize your implementation of OSA events for use by scripting components and how to create an **aete** resource.

Supporting OSA Events as a Client Application

Because users can send OSA events to a variety of applications simply by executing a script, many applications have no need to send OSA events. However, if you want to factor your application for recording, or if you want your application to send OSA events directly to other applications, you can use OSA Event Manager routines to create and send OSA events.

To send an OSA event, you must:

- Create the OSA event

- Add parameters and attributes
- Use the AESend function to send the event

Creating and Sending OSA Events describes how to perform these tasks.

Supporting OSA Event Objects

If your application responds to core and functional-area OSA events, it must also be able to resolve object specifier records that describe the objects on which those OSA events can act. In addition to the tasks described in Responding to OSA Events, you must perform the following tasks to handle OSA events that contain object specifier records:

- Write object accessor functions that can locate the OSA event objects you support, and install corresponding entries in your application's object accessor dispatch table.
- Write any object callback functions that you decide to provide. To handle object specifier records that specify a test, your application must provide at least two object callback functions: one that counts objects and one that compares them.
- Call AEResolve from your OSA event handlers whenever an OSA event parameter includes an object specifier record.

Resolving and Creating Object Specifier Records describes how to perform these tasks. It also describes how applications that send OSA events to themselves or directly to other applications can create object specifier records.

Supporting OSA Event Recording

If you make your application scriptable, you may also want to make it recordable. Users of recordable applications can record their actions in the form of OSA events that a scripting component translates into a script. When a user executes a recorded script, the scripting component sends the same OSA events to the application in which they were recorded.

To make your application recordable, you should use OSA events to report user actions to the OSA Event Manager in terms of OSA events. One way to do this is to separate the code that implements your application's user interface from the code that actually performs work when the user manipulates the interface. This is called *factoring* your application. A factored application acts as both the client and server application for OSA events it sends to itself in response to user actions. When recording is turned on, the OSA Event Manager sends a copy of every event than an application sends to itself to the scripting component or other process that turned recording on.

Introduction to Scripting provides an overview of how to make your application both scriptable and recordable. Recording OSA Events describes how to factor your application for recording and explains the OSA Event Manager's recording mechanism.

Responding to OSA Events

This chapter describes how your application can use the OSA Event Manager to respond to OSA events. Your application must be able to respond to the four required OSA events. To be scriptable, or capable of responding to OSA events sent by scripting components, your application should handle the appropriate core and functional-area OSA events.

Before you read this chapter, you should be familiar with Interapplication Communication and OSA Events. You should also have a copy of the *OSA Event Registry: Standard Suites* available for reference.

This chapter provides the basic information you need to make your application capable of responding to OSA events. To respond to core and functional-area OSA events, your application must also be able to resolve object specifier records. You should read Resolving and Creating Object Specifier Records before you write OSA event handlers for events that can contain object specifier records.

The section Handling OSA Events describes how to:

- Accept and process OSA events
- Install entries in the OSA event dispatch tables
- Handle the required events
- Get data out of an OSA event
- Write handlers that perform the action requested by an OSA event
- Reply to an OSA event
- Dispose of OSA event data structures
- Write and install coercion handlers

Handling OSA Events

You do not need to implement all OSA events at once. If you want to begin by supporting only the required OSA events, you must:

- Include code to handle OSA events in a window procedure.
- Write routines that handle the required events
- Install entries for the required OSA events in your application's OSA event dispatch table

The following sections explain how to perform these tasks: Accepting an OSA Event and Installing Entries in the OSA Event Dispatch Tables.

To respond to core and functional-area OSA events, you must install entries and write handlers for those events. You must also make sure that your application can locate OSA event objects with the aid of the OSA Event Manager routines described in Resolving and Creating Object Specifier Records. These routines are currently available as the Object Support Library (OSL).

Accepting an OSA Event

To accept or send OSA events, code must be included to handle OSA events in your application's main event loop.

The following code fragment is an example of a PM window procedure. This procedure receives the OSA event in the semantic event record specified in the *mp1* (message parameter 1). This procedure calls AEProcessOSAEvent function and passes the semantic event record found in *mp1* . The AEProcessOSAEvent function determines the type of OSA event received and calls the corresponding OSA event handler in your application.

```
MRESULT EXPENTRY MyObjectWindowProc(HWND hwnd, ULONG msg, MPARAM mp1, MPARA
{
  OSErr   rc = noErr;

  switch (msg)
  {
      case WM_SEMANTICEVENT:
        /* The pointer to the SemanticEvent is in mp1 */
        rc = AEProcessOSAEvent((SemanticEvent *)mp1);
        return ((MRESULT) rc);
      default:
        return (WinDefWindowProc(hwnd, msg, mp1, mp2));
        break;

  }       /* End of Switch */
}         /* End of WinProc */
```

You should provide an OSA event handler for each OSA event that your application supports. This handler is responsible for performing the action requested by the OSA event and if necessary can return data in the reply OSA event.

If the client application requests a reply, the OSA Event Manager passes a default reply OSA event to your handler. If the client application does not request a reply, the OSA Event Manager passes a null descriptor record (a descriptor record of descriptor type typeNull and a data handle whose value is NULL) to your handler instead of a default reply OSA event.

After your handler finishes processing the OSA event and adds any parameters to the reply OSA event, it must return a result code to AEProcessOSAEvent. If the client application is waiting for a reply, the OSA event Manager returns the reply OSA event to the client.

Installing Entries in the OSA Event Dispatch Tables

When your application receives a WM_SEMANTICEVENT message, the AEProcessOSAEvent function, with message parameter 1 of the

WM_SEMANTICEVENT message, is used to retrieve the OSA event. This function also routes the OSA event to the appropriate event handler. The application supplies an OSA event dispatch table to map the OSA events supported by the application to the OSA event handlers provided by the application.

To install entries in your application's OSA event dispatch table, use the AEInstallEventHandler function. You usually install entries for all of the OSA events that your application accepts into your application's OSA event dispatch table.

To install an OSA event handler in your OSA event dispatch table, you must specify the following:

- Event class of the OSA event
- Event ID of the OSA event
- Address of the OSA event handler for the OSA event
- Reference constant

You provide this information to the AEInstallEventHandler function. In addition, you indicate whether the entry should be added to your application's OSA event dispatch table or to the system OSA event dispatch table.

The *system OSA event dispatch table* contains system OSA event handlers-handlers that are available to all applications and processes running on the same computer. The handlers in your application's OSA event dispatch table are available only to your application. If AEProcessOSAEvent cannot find a handler for the OSA event in your application's OSA event dispatch table, it looks in the system OSA event dispatch table for a handler (see Dispatching OSA Event for details). If it does not find a handler for the event, it returns the errAEEventNotHandled result code.

If you add a handler to the system OSA event dispatch table, the handler should reside in a DLL. If there was already an entry in the system OSA event dispatch table for the same event class and event ID, it is replaced unless you chain it to your system handler. See Creating and Managing the OSA Event Dispatch Tables for details.

Installing Entries for the Required OSA Events

The following code fragment illustrates how to add entries for the required OSA events to your application's OSA event dispatch table.

```
myErr = AEInstallEventHandler(kCoreEventClass, kAEOpenApplication,
                              MyHandleOApp, 0, FALSE);
if (myErr != noErr) DoError(myErr);

myErr = AEInstallEventHandler(kCoreEventClass, kAEOpenDocuments,
                              MyHandleODoc,0, FALSE);
if (myErr != noErr) DoError(myErr);

myErr = AEInstallEventHandler(kCoreEventClass, kAEPrintDocuments,
                              MyHandlePDoc, 0, FALSE);
if (myErr != noErr) DoError(myErr);

myErr = AEInstallEventHandler(kCoreEventClass, kAEQuitApplication,
                              MyHandleQuit, 0, FALSE);
if (myErr != noErr) DoError(myErr);

return (myErr);
```

The previous code fragment creates entries for all four required OSA events in the OSA event dispatch table. The first entry creates an entry for the Open Application event. The entry indicates the event class and event ID of the Open Application event, supplies the address of the handler for that event, and specifies 0 as the reference constant.

The OSA Event Manager passes the reference constant to your handler each time your handler is called. Your application can use this reference constant for any purpose. If your application does not use the reference constant, use 0 as the value.

The *isSysHandler* parameter to the AEInstallEventHandler function is a Boolean value that determines whether the entry is added to the system OSA event dispatch table or to your application's OSA event dispatch table. To add the entry to your application's OSA event dispatch table, use FALSE as the value of this parameter. If you specify TRUE, the entry is added to the system OSA event dispatch table. The previous code fragment adds entries to the application's OSA event dispatch table.

Dispatching OSA Event

In addition to the OSA event handler dispatch tables, applications can add entries to a *special handler dispatch table* for application or system-wide usage. These dispatch tables are used for various specialized handlers; for more information, see Creating and Managing the Special Handler Dispatch Tables.

When an application calls AEProcessOSAEvent, the function looks first in the application's special handler dispatch table for an entry that was installed with the constant keyPreDispatch. If the application's special handler dispatch table does not include such a handler or if the handler returns errAEEventNotHandled, the function looks in the application's OSA event dispatch table for an entry that matches the event class and event ID of the specified OSA event.

If the application's OSA event dispatch table does not include such a handler or if the handler returns errAEEventNotHandled, the AEProcessOSAEvent function looks in the system special handler dispatch table for an entry that was installed with the constant keyPreDispatch. If the system special handler dispatch table does not include such a handler or if the handler returns errAEEventNotHandled, the function looks in the system OSA event dispatch table for an entry that matches the event class and event ID of the specified OSA event.

If the system OSA event dispatch table does not include such a handler, the OSA Event Manager returns the result code errAEEventNotHandled to the server application and, if the client application is waiting for a reply, to the client application.

For any entry in your OSA event dispatch table, you can specify a wildcard value for the event class, event ID, or both. You specify a wildcard by supplying the typeWildCard constant when installing an entry into the OSA event dispatch table. A wildcard value matches all possible values. Wildcards make it possible to supply one OSA event handler that dispatches several related OSA events.

For example, if you specify an entry with the typeWildCard event class and the kAEOpenDocuments event ID, the OSA Event Manager dispatches OSA events of any event class with an event ID of kAEOpenDocuments to the handler for that entry.

If you specify an entry with the kCoreEventClass event class and the typeWildCard event ID, the OSA Event Manager dispatches OSA events of the kCoreEventClass event class with any event ID to the handler for that entry.

If you specify an entry with the typeWildCard event class and the typeWildCard event ID, the OSA Event Manager dispatches all OSA events of any event class and any event ID to the handler for that entry.

If an OSA event dispatch table contains one entry for an event class and a specific event ID, and also contains another entry that is identical except that it specifies a wildcard value for either the event class or the event ID, the OSA Event Manager dispatches the more specific entry. For example, if an OSA event dispatch table includes one entry that specifies the event class as kAECoreSuite and the event ID as kAEDelete, and another entry that specifies the event class as kAECoreSuite and the event ID as typeWildCard, the OSA Event Manager will dispatch the OSA event handler associated with the entry that specifies the event ID as kAEDelete.

Important: If your application sends OSA events to itself using a typePID address descriptor record set to kCurrentProcess, the OSA Event Manager jumps directly to the appropriate OSA event handler without going through the normal event-processing sequence. This improves the performance of your application when sending events to yourself. For more information, see Addressing an OSA Event for Direct Dispatching.

Required OSA Events

If your application accepts OSA events, it must be able to process the four required OSA events.

Upon receiving any of the required OSA events, your application should perform the action requested by the event. Here is a summary of the contents of the required events and the actions they request applications to perform:

Open Application-perform tasks associated with opening an application

Event class	kCoreEventClass
Event ID	kAEOpenApplication
Parameters	None
Requested action	Perform any tasks (such as opening an untitled document window) that you would normally perform when a user opens your application without opening or printing any documents.

Open Documents-open the specified documents

Event class	kCoreEventClass
Event ID	kAEOpenDocuments
Required parameter	
Keyword:	keyDirectObject

Descriptor type:	typeAEList
Data:	A list of file names for the documents to be opened
Requested action	Open the documents specified in the keyDirectObject parameter.

Print Documents-print the specified documents

Event class	kCoreEventClass
Event ID	kAEPrintDocuments
Required parameter	
Keyword:	keyDirectObject
Descriptor type:	typeAEList
Data:	A list of file names for the documents to be printed
Requested action	Print the documents specified in the keyDirectObject parameter without opening windows for the documents.

Quit Application-perform tasks associated with quitting

Event class	kCoreEventClass
Event ID	kAEQuitApplication
Parameters	None
Requested action	Perform any tasks that your application would normally perform when the user chooses Quit. Such tasks typically include asking the user whether to save documents that have been changed.

Your application needs to recognize only two descriptor types to handle the required OSA events: descriptor lists and file names. The Open Documents event and Print Documents event use descriptor lists to store a list of documents to open. Each document is specified as a file name in the descriptor list.

Handling the Open Application Event

On startup, your application typically performs any needed initialization, and then begins to process events. To handle the Open Application event, your application should do just what the user expects it to do when it is opened. For example, your application might open a new untitled window in response to an Open Application event.

The following code fragment shows a handler that processes the Open Application event. This handler first calls an application-defined function called MyGotRequiredParams, which checks whether the OSA event contains any required parameters. If so, the handler returns an error, because by definition, the Open Application event should not contain any required parameters. Otherwise, the handler opens a new document window.

```
OSErr MyHandleOApp(OSAEvent theOSAEvent, OSAEvent reply, long handlerRefcon
{
  OSErr myErr;

  myErr = MyGotRequiredParams(theOSAEvent);
  if (myErr == noErr)
      DoNew;
  return (myErr);
}
```

For a description of the MyGotRequiredParams function, see the code fragment in section Writing OSA Event Handlers. For information about the *reply* and *handlerRefcon* parameters for an OSA event handler, see Writing OSA Event Handlers.

Handling the Open Documents Event

To handle the Open Documents event, your application should open the documents that the Open Documents event specifies in its direct parameter. Your application extracts this information and then opens the specified documents. The following code fragment shows a handler for the Open Documents event.

```
OSErr MyHandleODoc (OSAEvent theOSAEvent, OSAEvent reply, long handlerRefcc
{
  char       myFileName;
  AEDescList docList;
  OSErr       myErr, ignoreErr;
  long       index, itemsInList;
  Size       actualSize;
  AEKeyword   keywd;
  DescType   returnedType, typeOS2FileName;

        /* get the direct parameter--a descriptor list--and put it */
        /* into docList */
  myErr = AEGetParamDesc(&theOSAEvent, keyDirectObject, typeAEList, &docLis

  if (myErr == noErr) {
        /*check for missing required parameters*/
        myErr = MyGotRequiredParams(theOSAEvent);
        if (myErr == noErr) {
            /*count the number of descriptor records in the list*/
            myErr = AECountItems (&docList, &itemsInList);
            if (myErr == noErr)
              /*now get each descriptor record from the list, */
              /* coerce the returned data to a character string, and */
              /* open the associated file*/
              for (index=1; index >= itemsInList; ++index) {
                  myErr = AEGetNthPtr(&docList, index, typeOS2FileName, &ke
                                      &returnedType, &myFileName,
                                      sizeof(myFileName), &actualSize);
                  if (myErr == noErr) {
                      myErr = MyOpenFile(&myFileName);
                      if (myErr != noErr)
                          ;           /* handle error from MyOpenFile */
                      else
                          ;         /* handle error from AEGetNthPtr */
                  }
                }
        }
        else
          ;                  /* handle error from MyGotRequiredParams */
        ignoreErr = AEDisposeDesc(&docList);
  }
  else
      ;               /* failed to get direct parameter, handle error */
  return (myErr);
}
```

The handler in the previous code fragment first uses the AEGetParamDesc function to get the direct parameter (specified by the keyDirectObject keyword) out of the OSA event. The handler requests that AEGetParamDesc return a descriptor list in the *docList* variable. The handler then checks that it has retrieved all of the required parameters by calling the MyGotRequiredParams function. (See the code fragment in section Writing OSA Event Handlers for a description of this function.)

Once the handler has retrieved the descriptor list from the OSA event, it uses AECountItems to count the number of descriptors in the list. Using the returned number as an index, the handler can get the data of each descriptor record in the list. This handler requests that the

AEGetKeyPtr function coerce the data in the descriptor record to a file name. The handler can then use the file name as a parameter to its own routine for opening files.

For more information on the AEGetKeyPtr and AECountItems functions, see Getting Data Out of a Descriptor List.

After extracting the file name that describes the document to open, your application can use this name to open the file. For example, in the previous code fragment, the code passes the file name to its routine for opening files, the MyOpenFile function.

The MyOpenFile function should be designed so that it can be called in response to both the Open Documents event and to events generated by the user. For example, when the user chooses **Open** from the **File** menu, the code that handles the mouse-down event uses an Open File dialog to let the user choose a file; it then calls MyOpenFile, passing the file name record returned by the dialog box. By isolating code that performs a requested action from code that interacts with the user, you can easily adapt your application to handle OSA events that request the same action.

Note the use of the AEDisposeDesc function to dispose of the descriptor list when your handler no longer requires the data in it. Your handler should also return a result code.

Handling the Print Documents Event

To handle the Print Documents event, your application should extract information about the documents to be printed from the direct parameter, then print the specified documents.

The handler for the Print Documents event shown in the following code fragment is similar to the handler for the Open Documents event, except that it prints the documents referred to in the direct parameter.

```
OSErr MyHandlePDoc (OSAEvent theOSAEvent, OSAEvent reply, long handlerRefcc
{
  char       myFileName;
  AEDescList docList;
  OSErr       myErr, ignoreErr;
  long       index, itemsInList;
  Size       actualSize;
  AEKeyword   keywd;
  DescType   returnedType, typeOS2FileName;

  /* get the direct parameter--a descriptor list--and put it */
  /* into docList */
  myErr = AEGetParamDesc(&theOSAEvent, keyDirectObject, typeAEList, &docLis
  if (myErr == noErr) {
      /*check for missing required parameters*/
      myErr = MyGotRequiredParams(theOSAEvent);
      if (myErr == noErr) {
        /*count the number of descriptor records in the list*/
        myErr = AECountItems (&docList, &itemsInList);
        if (myErr == noErr)
            /*now get each descriptor record from the list, */
            /* coerce the returned data to a character string, and */
            /* print the associated file*/
            for (index=1; index >= itemsInList; ++index) {
              myErr = AEGetNthPtr(&docList, index, typeOS2FileName, &keywd,
                                  &returnedType, &myFileName,
                                  sizeof(myFileName), &actualSize);
              if (myErr == noErr) {
                  myErr = MyPrintFile(&myFileName);
                  if (myErr != noErr)
                      ;               /* handle error from MyOpenFile */
              }
              else
```

```
                ;                       /* handle error from AEGetNthPtr */
            }
        }
        else
          ;                     /* handle error from MyGotRequiredParams */
        ignoreErr = AEDisposeDesc(&docList);
    }
    else
        ;               /* failed to get direct parameter, handle error */
    return (myErr);
}
```

Handling the Quit Application Event

To handle the Quit Application event, your application should take any actions that are necessary before it is terminated (such as saving any open documents). The following code fragment shows an example of a handler for the Quit Application event.

```
OSErr MyHandleQuit (OSAEvent theOSAEvent, OSAEvent reply, long handlerRefcc
{
    OSErr myErr;
    BOOL   userCanceled;

    /* check for missing required parameters */
    myErr = MyGotRequiredParams(theOSAEvent);
    if (myErr == noErr) {
        userCanceled = MyPrepareToTerminate;
        if (userCanceled)
          MyHandleQuit = kUserCanceled;
        else
          MyHandleQuit = noErr;
    }
    else
        return (myErr);
}
```

The handler in the previous code fragment calls another function supplied by the application, the MyPrepareToTerminate function. This function saves the documents for any open windows and returns a Boolean value that indicates whether the user canceled the Quit operation. This is another example of isolating code for interacting with the user from the code that performs the requested action. By structuring your application in this way, you can use the same routine to respond to a user action (such as choosing the **Quit** command from the **File** menu) or to the corresponding OSA event. (For a description of the MyGotRequiredParams function, see Writing OSA Event Handlers.)

Getting Data Out of an OSA Event

The OSA Event Manager stores the parameters and attributes of an OSA event in a format that is internal to the OSA Event Manager. You use OSA Event Manager functions to retrieve the data from an OSA event and return it to your application in a format your application can use.

Most of the functions that retrieve data from OSA event parameters and attributes are available in two forms: one that returns the desired data in a specified buffer and one that returns a descriptor record containing the same data. For example, the AEGetParamPtr function uses a specified buffer to return the data contained in an OSA event parameter, and the AEGetParamDesc function returns the descriptor record for a specified parameter.

You can also use OSA Event Manager functions to get data out of descriptor records, descriptor lists, and AE records. You use similar

functions to put data into descriptor records, descriptor lists, and AE records.

When your handler receives an OSA event, you typically use the AEGetParamPtr, AEGetAttributePtr, AEGetParamDesc, or AEGetAttributeDesc function to get the data out of the OSA event.

Some OSA Event Manager functions let your application request that the data be returned using any descriptor type, even if it is different from the original descriptor type. If the original data is of a different descriptor type, the OSA Event Manager attempts to coerce the data to the requested descriptor type.

For example, the AEGetParamPtr function lets you specify the desired descriptor type of the resulting data as follows:

```
OSAEvent theOSAEvent;
DescType returnedType;
LongInt   multResult;
Size     actualSize;
OSErr     myErr;

myErr = AEGetParamPtr(&theOSAEvent, keyMultResult, typeLongInteger,
                      &returnedType, &multResult, sizeof(multResult),
                      &actualSize);
```

In this example, the desired type is specified in the third parameter by the typeLongInteger descriptor type. This requests that the OSA Event Manager coerce the data to a long integer if it is not already of this type. To prevent coercion and ensure that the descriptor type of the result is of the same type as the original, specify typeWildCard for the third parameter.

The OSA Event Manager returns, in the *returnedType* parameter, the descriptor type of the resulting data. This is useful information when you specify typeWildCard as the desired descriptor type; you can determine the descriptor type of the resulting data by examining this parameter.

The OSA Event Manager can coerce many different types of data. For example, the OSA Event Manager can convert integers to Boolean data types, and characters to numeric data types, in addition to other data type conversions. For a complete list of the data types for which the OSA Event Manager provides coercion handling, see the table in section Writing and Installing Coercion Handlers.

To perform data coercions that the OSA Event Manager does not perform, you can provide your own coercion handlers. See Writing and Installing Coercion Handlers for information on providing your own coercion handlers.

OSA event parameters are keyword-specified descriptor records. You can use AEGetParamDesc to get the descriptor record of a parameter, or you can use AEGetParamPtr to get the data out of the descriptor record of a parameter. If an OSA event parameter consists of an object specifier record, you can use AEResolve and your own object accessor functions to resolve the object specifier record-that is, to locate the OSA event object it describes. For more information about AEResolve and object accessor functions, see Writing Object Accessor Functions. Attributes are also keyword-specified descriptor records, and you can use similar routines to get the descriptor record of an attribute or to get the data out of an attribute.

The following sections show how to use the AEGetParamPtr, AEGetAttributePtr, AEGetParamDesc, or AEGetAttributeDesc function to get the data out of an OSA event.

Getting Data Out of an OSA Event Parameter

You can use the AEGetParamPtr or AEGetParamDesc function to get the data out of an OSA event parameter. Use the AEGetParamPtr function (or the AEGetKeyPtr function, which works the same way) to return the data contained in a parameter. Use the AEGetParamDesc function when you need to get the descriptor record of a parameter or to extract the descriptor list from a parameter.

For example, suppose you need to get the data out of a Set Data event. The keyAEData parameter of the OSA event contains the new data for the object. You can use the AEGetParamPtr function to get the data out of the OSA event.

You specify the OSA event that contains the desired parameter, the keyword of the desired parameter, the descriptor type the function should use to return the data, a buffer to store the data, and the size of this buffer as parameters to the AEGetParamPtr function. The AEGetParamPtr function returns the descriptor type of the resulting data and the actual size of the data, and it places the requested data in the specified buffer.

```
long           theInteger;
OSAEvent       theOSAEvent;
DescType       returnedType;
```

```
Size            actualSize;
OSErr           myErr;

myErr = AEGetParamPtr(&theOSAEvent, keyAEData, typeLongInteger,
                      &returnedType, &theInteger, sizeof(theInteger),
                      &actualSize);
```

In this example, the keyAEData keyword specifies the parameter from which the data should be extracted; AEGetParamPtr returns the data in the buffer specified by the *theInteger* variable.

You can request that the OSA Event Manager return the data using the descriptor type of the original data or you can request that the OSA Event Manager coerce the data into a descriptor type that is different from the original. To prevent coercion, specify the desired descriptor type as typeWildCard.

In this example, the AEGetParamPtr function returns, in the *returnedType* variable, the descriptor type of the resulting data. The descriptor type of the resulting data matches the requested descriptor type unless the OSA Event Manager was not able to coerce the data to the specified descriptor type or you specified the desired descriptor type as typeWildCard. If the coercion fails, the OSA Event Manager returns the errAECoercionFail result code.

The AEGetParamPtr function returns, in the *actualSize* variable, the actual size of the data (that is, the size of coerced data, if any coercion was performed). If the value returned in this variable is greater than the amount your application allocated for the buffer to hold the returned data, your application can increase the size of its buffer to this amount, and get the data again. You can also choose to use the AEGetParamDesc function when your application does not know the size of the data.

In general, use the AEGetParamPtr function to extract data that is of fixed length or known maximum length, and the AEGetParamDesc function to extract data that is of variable length. The AEGetParamDesc function returns the descriptor record for an OSA event parameter. This function is useful, for example, for extracting a descriptor list from a parameter.

You specify, as parameters to AEGetParamDesc, the OSA event that contains the desired parameter, the keyword of the desired parameter, the descriptor type the function should use to return the descriptor record, and a buffer to store the returned descriptor record. The AEGetParamDesc function returns the descriptor record using the specified descriptor type.

For example, the direct parameter of the Open Documents event contains a descriptor list that specifies the documents to open. You can use the AEGetParamDesc function to get the descriptor list out of the direct parameter.

```
AEDescList docList;
OSAEvent   theOSAEvent;
OSErr       myErr;

myErr = AEGetParamDesc(&theOSAEvent, keyDirectObject, typeAEList, &docList)
```

In this example, the OSA event specified by the *theOSAEvent* variable contains the desired parameter. The keyDirectObject keyword specifies that the AEGetParamDesc function should get the descriptor record of the direct parameter. The typeAEList descriptor type specifies that the descriptor record should be returned as a descriptor list. In this example, the AEGetParamDesc function returns a descriptor list in the *docList* variable.

The descriptor list contains a list of descriptor records. To get the descriptor records and their data out of a descriptor list, use the AECountItems function to find the number of descriptor records in the list and then make repetitive calls to the AEGetKeyPtr function to get the data out of each descriptor record. See Getting Data Out of a Descriptor List for more information.

Note that the AEGetParamDesc function copies the descriptor record from the parameter. When you are done with a descriptor record that you obtained from AEGetParamDesc, you must dispose of it by calling the AEDisposeDesc function.

If an OSA event parameter consists of an object specifier record, you can use AEResolve to resolve the object specifier record (that is, locate the OSA event object it describes), as explained in Finding OSA Event Objects.

Getting Data Out of an Attribute

You can use the AEGetAttributePtr or AEGetAttributeDesc function to get the data out of the attributes of an OSA event.

You specify, as parameters to AEGetAttributePtr, the OSA event that contains the desired attribute, the keyword of the desired attribute, the descriptor type the function should use to return the data, a buffer to store the data, and the size of this buffer. The AEGetAttributePtr function

returns the descriptor type of the returned data and the actual size of the data and places the requested data in the specified buffer.

For example, this code gets the data out of the keyEventSourceAttr attribute of an OSA event.

```
OSAEvent theOSAEvent;
DescType returnedType;
Integer   sourceOfAE;
Size     actualSize;
OSErr     myErr;

myErr = AEGetAttributePtr(theOSAEvent, keyEventSourceAttr, typeShortInteger
                          returnedType, &sourceOfAE, SizeOf(sourceOfAE),
                          actualSize);
```

The keyEventSourceAttr keyword specifies the attribute from which to get the data. The typeShortInteger descriptor type specifies that the data should be returned as a short integer; the *returnedType* variable contains the actual descriptor type that is returned. You also must specify a buffer to hold the returned data and specify the size of this buffer. If the data is not already a short integer, the OSA Event Manager coerces it as necessary before returning it. The AEGetAttributePtr function returns, in the *actualSize* variable, the actual size of the returned data after coercion has taken place. You can check this value to make sure you got all the data.

As with the AEGetParamPtr function, you can request that AEGetAttributePtr return the data using the descriptor type of the original data, or you can request that the OSA Event Manager coerce the data into a descriptor type that is different from the original.

In this example, the AEGetAttributePtr function returns the requested data as a short integer in the *sourceOfAE* variable, and you can get information about the source of the OSA event by examining this value. You can test the returned value against the values defined by the data type AEEventSource.

```
typedef unsigned char AEEventSource;
```

The constants defined by the data type AEEventSource. have the following meanings:

Constant	Meaning
kAEUnknownSource	Source of OSA event is unknown.
kAEDirectCall	A direct call that bypassed the PM message queue.
kAESameProcess	Target application is also the source application.
kAELocalProcess	Source application is another process on the same computer as the target application.

The next example shows how to use the AEGetAttributePtr function to get data out of the keyMissedKeywordAttr attribute. After your handler extracts all known parameters from an OSA event, it should check whether the keyMissedKeywordAttr attribute exists. If it does, then your handler did not get all of the required parameters.

Note that if AEGetAttributePtr returns the errAEDescNotFound result code, then the keyMissedKeywordAttr attribute does not exist-that is, your application has extracted all of the required parameters. If AEGetAttributePtr returns noErr, then the keyMissedKeywordAttr attribute does exist-that is, your handler did not get all of the required parameters.

```
OSAEvent theOSAEvent;
DescType returnedType;
Size     actualSize;
OSErr     myErr;

myErr = AEGetAttributePtr(&theOSAEvent, keyMissedKeywordAttr, typeWildCard,
                          returnedType, NULL, 0, &actualSize);
```

The data in the keyMissedKeywordAttr attribute contains the keyword of the first required parameter, if any, that your handler did not retrieve. If you want this data returned, specify a buffer to hold it and specify the buffer size. Otherwise, as in this example, specify NULL as the buffer and 0 as the size of the buffer.

This example shows how to use the AEGetAttributePtr function to get the address of the sender of an OSA event from the keyAddressAttr attribute of the OSA event:

```
OSAEvent theOSAEvent;
DescType returnedType;
PID       addressOfAE;
Size     actualSize;
OSErr     myErr;

myErr = AEGetAttributePtr(&theOSAEvent, keyAddressAttr, typePID,
                          &returnedType, &addressOfAE, sizeof(PID),
                          &actualSize);
```

For more information about target addresses, see Specifying a Target Address.

Getting Data Out of a Descriptor List

You can use the AECountItems function to count the number of items in a descriptor list, and you can use AEGetKeyDesc or AEGetKeyPtr to get a descriptor record or its data out of a descriptor list.

The Open Documents event contains a direct parameter that specifies the list of documents to open. The list of documents is contained in a descriptor list. After extracting the descriptor list from the parameter, you can determine the number of items in the list and then extract each descriptor record from the descriptor list. See the second figure in section Descriptor Lists for a depiction of the Open Documents event.

For example, when your handler receives an Open Documents event, you can use the AEGetParamDesc function to return the direct parameter as a descriptor list. You can then use AECountItems to return the number of descriptor records in the list.

```
OSAEvent   theOSAEvent;
AEDescList docList;
LongInt     itemsInList;
OSErr       myErr;

myErr = AEGetParamDesc(&theOSAEvent, keyDirectObject, typeAEList, &docList)
myErr = AECountItems(&docList, &itemsInList);
```

The AEGetParamDesc function returns, in the *docList* variable, a copy of the descriptor list from the direct parameter of the Open Documents event. You specify this list to the AECountItems function.

You specify the descriptor list whose items you want to count in the *theAEDescList* parameter to AECountItems. The OSA Event Manager returns, in the *theCount* parameter, the number of items in the list. When extracting the descriptor records from a list, you often use the number of items as a loop index. Here is an example:

```
for (index = 1; index<=itemsInList; ++index) {
    /* for each descriptor record in the list, get its data */
    ...
}
```

The format of the descriptor records in a descriptor list is private to the OSA Event Manager. You must use the AEGetKeyPtr or AEGetKeyDesc function to extract descriptor records from a descriptor list.

You specify the descriptor list that contains the desired descriptor records and an index as parameters to the AEGetKeyPtr function. The index represents a specific descriptor record in the descriptor list. The AEGetKeyPtr function returns the data for the descriptor record represented by the specified index.

You also specify the descriptor type the function should use to return the data, a buffer to store the data, and the size of this buffer. If the specified descriptor record exists, the AEGetKeyPtr function for the descriptor record represented by the specified index. returns the keyword of the parameter, the descriptor type of the returned data, and the actual size of the data, and it places the requested data in the specified buffer.

Here is an example that uses the AEGetKeyPtr function to extract an item from the descriptor list in the direct parameter of the Open Documents event:

```
AEDescList docList;
long       index;
AEKeyword   keywd;
DescType   returnedType;
char       myFileName;
Size       actualSize;
OSErr       myErr;

myErr = AEGetNthPtr(&docList, index, typeOS2FileName, keywd, &returnedType,
                    &myFileName, sizeof(myFileName), &actualSize);
```

The *docList* variable specifies the descriptor list from the direct parameter of the Open Documents event. The *index* variable specifies the index of the descriptor record to extract. You can use the typeOS2FileName descriptor type, as in this example, to specify that the data be returned as a path name. The AEGetKeyPtr function returns the keyword for the descriptor record represented by the specified index of the parameter and the descriptor type of the resulting data in the *keywd* and *returnedType* variables, respectively.

You also specify a buffer to hold the desired data and the size (in bytes) of the buffer. In this example, the *myFileName* variable specifies the buffer. The function returns the actual size of the data in the *actualSize* variable. If this size is larger than the size of the buffer you provided, you know that you did not get all of the data for the descriptor record.

The following code fragment shows a more complete example of extracting the items from a descriptor list in the Open Documents event.

```
long       index, itemsInList;
OSErr       myErr;
AEDescList docList;
AEKeyword   keywd;
DescType   returnedType, typeOS2FileName;
char       myFileName;
Size       actualSize;

for (index=1; index >= itemsInList; ++index) {
    myErr= AEGetNthPtr(&docList, index, typeOS2FileName, &keywd, &returnedT
                       &myFileName, sizeof(myFileName), &actualSize);
    if (myErr =! noErr)
      DoError(myErr);
    myErr = MyOpenFile(&myFileName);
    if (myErr =! noErr)
      DoError(myErr);
}
myErr = AEDisposeDesc(&docList);
```

Writing OSA Event Handlers

For each OSA event your application supports, you must provide a function called an OSA event handler. The AEProcessOSAEvent function calls one of your OSA event handlers when it processes an OSA event. Your OSA event handlers should perform any action requested by the OSA event, add parameters to the reply OSA event if appropriate, and return a result code.

The OSA Event Manager uses dispatch tables to route OSA events to the appropriate OSA event handler. You must supply an OSA event handler for each entry in your application's OSA event dispatch table. Each handler must be a function that uses this syntax:

```
OSErr APIENTRY MyEventHandler (OSAEvent *theOSAEvent, OSAEvent reply,
                               long handlerRefcon);
```

The *theOSAEvent* parameter is the OSA event to handle. Your handler uses OSA Event Manager functions to extract any parameters and attributes from the OSA event and then performs the necessary processing. If any of the parameters include object specifier records, your handler should call AEResolve to resolve them-that is, to locate the OSA event objects they describe. For more information, see Resolving and Creating Object Specifier Records in this book.

The *reply* parameter is the default reply provided by the OSA Event Manager. (Replying to an OSA Event describes how to add parameters to the default reply.) The *handlerRefcon* parameter is the reference constant stored in the OSA event dispatch table entry for the OSA event. Your handler can check the reference constant, if necessary, for information about the OSA event.

You can use the reference constant for anything you wish. For example, if you want to use the same handler for several OSA events, you can install entries for each event in your application's OSA event dispatch table that specify the same handler but different reference constants. Your handler can then use the reference constant to distinguish the different OSA events it handles.

After extracting all known parameters from the OSA event, every handler should determine whether the OSA event contains any further required parameters. Your handler can determine whether it retrieved all the required parameters by checking whether the keyMissedKeywordAttr attribute exists. If the attribute exists, then your handler has not retrieved all the required parameters and should immediately return an error. If the attribute does not exist, then the OSA event does not contain any more required parameters, although it may contain additional optional parameters.

The OSA Event Manager determines which parameters are optional according to the keywords listed in the keyOptionalKeywordAttr attribute. The source application is responsible for adding these keywords to the keyOptionalKeywordAttr attribute, but is not required to do so, even if that parameter is listed in the *OSA Event Registry: Standard Suite*s as an optional parameter. If the source application does not add the necessary keyword to the keyOptionalKeywordAttr attribute, the target application treats the parameter as required for that OSA event. If the target application supports the parameter, it should handle the OSA event as the source application expects. If the target application does not support the parameter and checks whether it has received all the required parameters, it finds that there is another parameter that the client application considered required, and should return the result code errAEParamMissed without attempting to handle the event.

The following code fragment shows a function that checks for a keyMissedKeywordAttr attribute. A handler calls this function after getting all the required parameters it knows about from an OSA event.

```
OSErr MyGotRequiredParams (OSAEvent theOSAEvent)
{
  OSErr     myErr;
  DescType   desiredType, *descriptorType;
  AEKeyword theAEKeyword;
  Size       actualSize;

  myErr = AEGetAttributePtr(&theOSAEvent, theAEKeyword, desiredType,
                            descriptorType, NULL, 0, &actualSize);
  if (myErr == errAEDescNotFound)
      /*you got all the required parameters*/
      return (noErr);
  if (myErr == noErr)
      /*you missed a required parameter*/
      return (errAEParamMissed);
}
```

The code in the previous code fragment uses the AEGetAttributePtr function to get the keyMissedKeywordAttr attribute. This attribute contains the first required parameter, if any, that your handler did not retrieve. If AEGetAttributePtr returns the errAEDescNotFound result code, the OSA event does not contain a keyMissedKeywordAttr attribute. If the OSA event does not contain this attribute, then your handler has extracted all of the parameters that the client application considered required.

If the AEGetAttributePtr function returns noErr as the result code, then the attribute does exist, meaning that your handler has not extracted all of the required parameters. In this case, your handler should return an error and not process the OSA event.

The first remaining required parameter is specified by the data of the keyMissedKeywordAttr attribute. If you want this data returned, specify a buffer to hold the data. Otherwise, specify NULL as the buffer and 0 as the size of the buffer. If you specify a buffer to hold the data, you can check the value of the *actualSize* parameter to see if the data is larger than the buffer you allocated.

For more information about specifying OSA event parameters as optional or required, see Specifying Optional Parameters for an OSA Event.

Replying to an OSA Event

Your handler routine for a particular OSA event is responsible for performing the action requested by the OSA event, and can optionally return data in a reply OSA event. The OSA Event Manager passes a default reply OSA event to your handler. The default reply OSA event has no parameters when it is passed to your handler. Your handler can add parameters to the reply OSA event. If the client application requested a reply, the OSA Event Manager returns the reply OSA event to the client.

The reply OSA event is identified by the kCoreEventClass event class and by the kAEAnswer event ID. If the client application specified the kAENoReply flag in the *reply* parameter of the AESend function, the OSA Event Manager passes a null descriptor record (a descriptor record of type typeNull whose data handle has the value NULL) to your handler instead of a default reply OSA event. Your handler should check the descriptor type of the reply OSA event before attempting to add any attributes or parameters to it. An attempt to add an OSA event attribute or parameter to a null descriptor record generates an error.

If the client application requests a reply, the OSA Event Manager prepares a reply OSA event for the client by passing a default reply OSA event to your handler. The default reply OSA event has no parameters when it is passed to your handler. Your handler can add any parameters to the reply OSA event. If your application is a spelling checker, for example, you can return a list of misspelled words in a parameter.

When your handler finishes processing an OSA event, it returns a result code to AEProcessOSAEvent, which returns this result code as its function result. If your handler returns a nonzero result code, and if you have not added your own keyErrorNumber parameter, the OSA Event Manager also returns this result code to the client application by putting the result code into a keyErrorNumber parameter for the reply OSA event. The client can check for the existence of this parameter to determine whether the handler performed the requested action.

The client application specifies whether it wants a reply OSA event or not by specifying flags (represented by constants) in the *sendMode* parameter of the AESend function.

If the client specifies the kAEWaitReply flag in the *sendMode* parameter, the AESend function does not return until the timeout specified by the *timeOutInTicks* parameter expires or the server application returns a reply. When the server application returns a reply, the *reply* parameter to AESend contains the reply OSA event that your handler returned to the AEProcessOSAEvent function. When the client application no longer needs the original OSA event and the reply event, it must dispose of them, but the OSA Event Manager disposes of both the OSA event and the reply event for the server application when the server's handler returns to AEProcessOSAEvent.

If the client specified the kAEQueueReply flag, the client receives the reply event at a later time during its normal processing of other events.

Your handler should always set its function result to noErr if it successfully handles the OSA event. If an error occurs, your handler should return either errAEEventNotHandled or some other nonzero result code. If the error occurs because your application cannot understand the event, return errAEEventNotHandled. This allows the OSA Event Manager to look for a handler in the system special handler or system OSA event dispatch tables that might be able to handle the event. If the error occurs because the event is impossible to handle as specified, return the result code returned by whatever function caused the failure, or whatever other result code is appropriate.

For example, suppose your application receives a Get Data event requesting the name of the current printer, and your application cannot handle such an event. In this situation, you should return errAEEventNotHandled in case another handler available to the OSA Event Manager can handle the Get Data event. This strategy allows users to take advantage of system capabilities from within your application via system handlers.

However, if your application cannot handle a Get Data event that requests the fifth paragraph in a document because the document contains only four paragraphs, you should return some other nonzero error, because further attempts to handle the event are pointless.

If your OSA event handler calls the AEResolve function and AEResolve calls an object accessor function in the system object accessor dispatch table, your OSA event handler may not recognize the descriptor type of the token returned by the function. In this case, your handler should return the result code errAEUnknownObjectType. When your handler returns this result code, the OSA Event Manager attempts to locate a system OSA event handler that can recognize the token. For more information, see Installing Entries in the Object Accessor Dispatch Tables.

The OSA Event Manager automatically adds any nonzero result code that your handler returns to a keyErrorNumber parameter in the reply OSA event. In addition to returning a result code, your handler can also return an error string in the keyErrorString parameter of the reply OSA event. Your handler should provide meaningful text in the keyErrorString parameter, so that the client can display this string to the user if desired.

The following code fragment shows how to add the keyErrorString parameter to the reply OSA event. See Adding Parameters to an OSA Event for a description of the AEPutParamPtr function.

```
OSErr MyHandler (OSAEvent theOSAEvent, OSAEvent reply, long handlerRefcon)
{
  OSErr     myErr;
  char       errStr;
  DescType   typeOS2Text;
  LONG       size;

  /* handle your OSA event here */

  /* if an error occurs when handling an OSA event, set the */
```

```
  /* function result and error string accordingly */
  if (myErr != noErr) {
      return (myErr);      /* result code to be returned--the */
                           /* OSA Event Manager. adds this */
                           /* result code to the reply OSA event */
                           /* as the keyErrorNumber parameter */
/*    if (reply.dataHandle != NULL) { */
      myErr = AESizeOfDescData(&reply, &size);
      if (size != 0) {
      /* add error string parameter to the default reply */
        errStr = 'Why error occurred';
        myErr = AEPutParamPtr(&reply, keyErrorString, typeOS2Text,
                              &errStr, strlen(errStr));
      }
  }
  else
    return (noErr);
}
```

If your handler needs to return data to the client, it can add parameters to the reply OSA event. The following code fragment shows how a handler for the Multiply event (an imaginary OSA event that asks the server to multiply two numbers) might return the results of the multiplication to the client.

```
OSErr MyMultHandler (OSAEvent theOSAEvent, OSAEvent reply, long handlerRefc
{
  OSErr     myErr;
  long      number1, number2;
  long      replyResult;
  Size      actualSize;
  DescType returnedType;

  /* get the numbers to multiply from the parameters of the */
  /* OSA event; put the numbers in the number1 and number2 */
  /* variables and then perform the requested multiplication */
  myErr = MyDoMultiply(&theOSAEvent, number1, number2, &replyResult);
  if (myErr == noErr)
      if (reply.dataHandle != NULL)
          /* return result of the multiplication in the reply OSA event */
          myErr = AEPutParamPtr(&reply, keyDirectObject, typeLongInteger,
                                &replyResult, sizeof(replyResult));
  return (myErr);
  /* if an error occurs, set the error string accordingly, as shown in */
  /* the previous code example. */
}
```

Disposing of OSA Event Data Structures

Whenever a client application uses OSA Event Manager functions to create a descriptor record, descriptor list, or OSA event record, the OSA Event Manager allocates memory for these data structures in the client's application heap. Likewise, when a server application extracts a descriptor record from an OSA event by using OSA Event Manager functions, the OSA Event Manager creates a copy of the descriptor record, including the data to which its handle refers, in the server's application heap.

Whenever you finish using a descriptor record or descriptor list that you have created or extracted from an OSA event, you should dispose of the descriptor record-and thereby deallocate the memory it uses-by calling the AEDisposeDesc function. If the descriptor record you pass to

AEDisposeDesc (such as an OSA event record or an AE record) includes other nested descriptor records, one call to AEDisposeDesc will dispose of them all.

When a client application adds a descriptor record to an OSA event (for example, when it creates a descriptor record by calling AECreateDesc and then puts a copy of it into a parameter of an OSA event by calling AEPutParamDesc), it is still responsible for disposing of the original descriptor record. After a client application has finished using both the OSA event specified in the AESend function and the reply OSA event, it should dispose of their descriptor records by calling AEDisposeDesc. The client application should dispose of them even if AESend returns a nonzero result code.

The OSA event that a server application's handler receives is a copy of the original event created by the client application. When a server application's handler returns to AEProcessOSAEvent, the OSA Event Manager disposes of the server's copy (in the server's application heap) of both the OSA event and the reply event. The server application is responsible for disposing of any descriptor records created while extracting data from the OSA event or adding data to the reply event.

In general, outputs from OSA Event Manager functions are your application's responsibility. Once you finish using them, you should use AEDisposeDesc to dispose of any OSA event data structures created or returned by these functions:

AECoerceDesc	AEDuplicateDesc
AECoercePtr	AEGetAttributeDesc
AECreateOSAEvent	AEGetKeyDesc
AECreateDesc	AEGetKeyDesc
AECreateList	AEGetParamDesc

If you attempt to dispose of descriptor records returned by successful calls to these functions without using AEDisposeDesc, your application may not be compatible with future versions of the OSA Event Manager. However, if any of these functions return a nonzero result code, they return a null descriptor record, which does not need to be disposed of.

Outputs from functions, such as AEGetKeyPtr, that use a buffer rather than a descriptor record to return data do not require the use of AEDisposeDesc. It is therefore preferable to use these functions for any data that is not identified by a handle.

Some of the functions described in Resolving and Creating Object Specifier Records also create descriptor records. If you set the *dispose**Input**s* parameter to FALSE for any of the following functions, you should dispose of any OSA event data structures that they create or return:

CreateCompDescriptor	CreateObjSpecifier
CreateLogicalDescriptor	CreateRangeDescriptor

Your application is also responsible for disposing of some of the tokens it creates in the process of resolving an object specifier record. For information about token disposal, see Defining Tokens.

Writing and Installing Coercion Handlers

When your application extracts data from a parameter, it can request that the OSA Event Manager return the data using a descriptor type that is different from the original descriptor type. For example, when extracting data from the direct parameter of the Open Documents event, you can request that the character be returned as an integer. The OSA Event Manager can automatically coerce many different types of data from one to another. The table below shows descriptor types and the kinds of coercion that the OSA Event Manager can perform.

You can also provide your own routines, referred to as co*ercion handler*s , to coerce data into any other descriptor type. To install your own coercion handlers, use the AEInstallCoercionHandler function. You specify as parameters to this function a:

- Descriptor type of the data coerced by the handler
- Descriptor type of the resulting data
- Address of the coercion handler for this descriptor type
- Reference constant
- Boolean value that indicates whether your coercion handler expects the data to be specified as a descriptor record or as a pointer to the actual data
- Boolean value that indicates whether your coercion handler should be added to your application's coercion dispatch table or the system coercion dispatch table

The s*ystem coercion dispatch table* is a table in the DLL that contains coercion handlers available to all applications and processes running on the same computer. The coercion handlers in your application's coercion dispatch table are available only to your application. When

attempting to coerce data, the OSA Event Manager first looks for a coercion handler in your application's coercion dispatch table. If it cannot find a handler for the descriptor type, it looks in the system coercion dispatch table for a handler. If it does not find a handler there, it attempts to use the default coercion handling described by the table below. If it cannot find an appropriate default coercion handler, it returns the errAECoercionFail result code.

Any handler that you add to the system coercion dispatch table should reside in a DLL. If there was already an entry in the system coercion dispatch table for the same descriptor type, it is replaced. Therefore, if there is an entry in the system coercion dispatch table for the same descriptor type, you should chain it to your system coercion handler as explained in Creating and Managing the Coercion Handler Dispatch Tables.

You can provide a coercion handler that expects to receive the data in a descriptor record or a buffer referred to by a pointer. When you install your coercion handler, you specify how your handler wishes to receive the data. Whenever possible, you should write your coercion handler so that it can accept a pointer to the data, because it is more efficient for the OSA Event Manager to provide your coercion handler with a pointer to the data.

A coercion handler that accepts a pointer to data must be a function with the following syntax:

```
OSErr MyCoercePtr (DescType typeCode, void *dataPtr, Size dataSize,
                   DescType toType, long handlerRefcon, AEDesc *result);
```

The *typeCode* parameter is the descriptor type of the original data. The *dataPtr* parameter is a pointer to the data to coerce; the *dataSize* parameter is the length, in bytes, of the data. The *toType* parameter is the desired descriptor type of the resulting data. The *handlerRefcon* parameter is a reference constant stored in the coercion table entry for the handler and passed to the handler by the OSA Event Manager whenever the handler is called. The *result* parameter is the descriptor record returned by your coercion handler.

Your coercion handler should coerce the data to the desired descriptor type and return the data in the descriptor record specified by the *result* parameter. If your handler successfully performs the coercion, it should return the noErr result code; otherwise, it should return a nonzero result code.

A coercion handler that accepts a descriptor record must be a function with the following syntax:

```
OSErr MyCoerceDesc (AEDesc *theAEDesc, DescType toType,
                    long handlerRefcon, AEDesc *result,);
```

The *theAEDesc* parameter is the descriptor record that contains the data to be coerced. The *toType* parameter is the descriptor type of the resulting data. The *handlerRefcon* parameter is a reference constant stored in the coercion table entry for the handler and passed to the handler by the OSA Event Manager whenever the handler is called. The *result* parameter is the resulting descriptor record.

Your coercion handler should coerce the data in the descriptor record to the desired descriptor type and return the data in the descriptor record specified by the *result* parameter. Your handler should return an appropriate result code.

Note: To ensure that no coercion is performed and that the descriptor type of the result is of the same descriptor type as the original, specify typeWildCard for the desired type.

The following table lists the descriptor types for which the OSA Event Manager provides coercion.

Original descriptor type of data to be coerced	Desired descriptor type	Description
typeChar	typeInteger typeLongInteger typeShortInteger typeMagnitude	Any string that is a valid representation of a number can be coerced into an equivalent numeric value.
typeInteger typeLongInteger typeShortInteger typeMagnitude	typeChar	Any numeric descriptor type can be coerced into the equivalent text string.
typeInteger typeLongInteger typeShortInteger typeMagnitude	typeInteger typeLongInteger typeShortInteger typeMagnitude	Any numeric descriptor type can be coerced into any other numeric descriptor type.
typeChar	typeType typeEnumerated typeKeyword typeProperty	Any four-character string can be coerced to one of these descriptor types.

typeEnumerated typeKeyword typeProperty typeType	typeChar	Any of these descriptor types can be coerced to the equivalent text string.
typeOS2Text	typeChar	The result contains text only, without the code page or country code from the original descriptor record.
typeTrue	typeBoolean	The result is the Boolean value TRUE.
typeFalse	typeBoolean	The result is the Boolean value FALSE.
typeEnumerated	typeBoolean	The enumerated value true becomes the Boolean value TRUE. The enumerated value fals becomes the Boolean value FALSE.
typeBoolean	typeEnumerated	The Boolean value FALSE becomes the enumerated value fals. The Boolean value TRUE becomes the enumerated value true.
typeShortInteger	typeBoolean	A value of 1 becomes the Boolean value TRUE. A value of 0 becomes the Boolean value FALSE.
typeBoolean	typeShortInteger	A value of FALSE becomes 0. A value of TRUE becomes 1.
any descriptor type	typeAEList	A descriptor record is coerced into a descriptor list containing a single item.
typeAEList	*type of list item*	A descriptor list containing a single descriptor record is coerced into a descriptor record.

Note: Some of the descriptor types listed in this table are synonyms; for example, the constants typeInteger and typeLongInteger have the same four-character code, **gnol**.

Reference for Responding to OSA Events

This section describes the basic OSA Event Manager data structures and routines that your application can use to respond to OSA events. It also describes the syntax for application-defined OSA event handlers and coercion handlers that your application can provide for use by the OSA Event Manager.

For information about routines used to create and send OSA events, see Creating and Sending OSA Events. For information about routines and data structures used with object specifier records, see Resolving and Creating Object Specifier Records.

Data Structures Used by the OSA Event Manager

This section summarizes the major data structures used by the OSA Event Manager. For an overview of the relationships among these data structures, see Data Structures within OSA Events.

Descriptor Records and Related Data Structures

Descriptor records are the fundamental data structures from which OSA events are constructed. A descriptor record is a data structure of type AEDesc.

```
typedef struct _AEDesc {
DescType       descriptorType;
Handle         dataHandle;
} AEDesc;
```

Field	Description
descriptorType	A four-character string of type DescType that indicates the type of data being passed.
dataHandle	A handle to the data being passed.

The descriptor type is a structure of type DescType, which in turn is of data type unsigned long-that is, a four-character code. Constants, rather than these four-character codes, are usually used to refer to descriptor types. The following table lists the constants for the basic descriptor types used by the OSA Event Manager.

Descriptor type	Value	Description
typeAEList	list	List of descriptor records
typeAERecord	reco	List of keyword-specified descriptor records
typeBoolean	bool	Boolean value
typeChar	TEXT	Unterminated string
typeEnumerated	enum	Enumerated data
typeFalse	fals	FALSE Boolean value
typeInteger	gnol	32-bit integer
typeKeyword	keyw	OSA event keyword
typeLongInteger	gnol	32-bit integer
typeMagnitude	ngam	Unsigned 32-bit integer
typeNull	null	Nonexistent data (data handle is NULL)
typeOSAEvent	aevt	OSA event record
typeOS2FileName		File name
typePID	PID	Process ID
typeProperty	prop	OSA event property
typeShortInteger	rohs	16-bit integer
typeTrue	true	TRUE Boolean value
typeType	type	Four-character code for event class or event ID
typeWildCard	****	Matches any type

For information about descriptor records and descriptor types used with object specifier records, see Resolving and Creating Object Specifier Records.

OSA event attributes, OSA event parameters, object specifier records, tokens, and most of the other data structures used by the OSA Event Manager are constructed from one or more descriptor records. The OSA Event Manager identifies the various parts of an OSA event by means of keywords associated with the corresponding descriptor records. The AEKeyword data type is defined as a four-character code.

```
typedef unsigned long AEKeyword;
```

Constants are typically used for keywords. A keyword combined with a descriptor record forms a keyword-specified descriptor record, which is defined by a data structure of type AEKeyDesc.

```
typedef struct _AEKeyDesc {
AEKeyword       descKey;
AEDesc          descContent;
} AEKeyDesc;
```

Field	Description
descKey	A four-character code of type AEKeyword that identifies the data in the *descContent* field.
descContent	A descriptor record of type AEDesc.

Every OSA event includes an attribute that contains the address of the target application. A descriptor record that contains an application's address is called an address descriptor record, or AEAddressDesc.

```
typedef AEDesc AEAddressDesc;
```

Many OSA Event Manager functions take or return lists of descriptor records in a special descriptor record called a descriptor list. A descriptor list is a structure of data type AEDescList whose data consists of a list of other descriptor records.

```
typedef AEDesc AEDescList;
```

Other OSA Event Manager functions take or return lists of keyword-specified descriptor records in the form of an AE record. An AE record is a structure of data type AERecord whose data handle refers to a list of keyword-specified descriptor records.

```
typedef AEDescList AERecord;
```

The handle for a descriptor list of data type AERecord refers to a list of keyword-specified descriptor records that specify OSA event parameters; they cannot specify OSA event attributes.

Finally, a full-fledged OSA event, including both attributes and parameters, is an OSA event record, which is a structure of data type OSAEvent.

```
typedef AERecord OSAEvent;
```

The event class and event ID of an OSA event are specified in OSA Event Manager routines by structures of data types AEEventClass and AEEventID, respectively.

```
typedef unsigned long AEEventClass;

typedef unsigned long AEEventID;
```

For more information about descriptor records and the other data structures described in this section, see Data Structures within OSA Events.

With the exception of array data records, which are described in the next section, the other OSA Event Manager data structures used in responding to OSA events are described in the section Routines for Responding to OSA events under the descriptions of the routines that use them.

OSA Event Array Data Types

The AEGetArray function creates a C array that corresponds to an *OSA event array* in a descriptor list, and the AEPutArray function adds data specified in a buffer to a descriptor list as an OSA event array.

You can use the data type AEArrayType to define the type of OSA event array you want to add to or obtain from a descriptor list.

```
typedef unsigned char AEArrayType;
```

When your application adds an OSA event array to a descriptor list, it provides the data for an OSA event array in an array data record, which is defined by the data type AEArrayData.

```
union AEArrayData {
short            kAEDataArray[1];
char            kAEPackedArray[1];
Handle          kAEHandleArray[1];
AEDesc          kAEDescArray[1];
AEKeyDesc        kAEKeyDescArray[1];
} AEArrayData;
```

The type of array depends on the data for the array:

Array type	Description of OSA event array
kAEDataArray	Array items consist of data of the same size and same type, and are aligned on word boundaries.
kAEDescArray	Array items consist of descriptor records of different descriptor types with data of variable size.
kAEHandleArray	Array items consist of handles to data of variable size and the same type.
kAEKeyDescArray	Array items consist of keyword-specified descriptor records with different keywords, different descriptor types, and data of variable size.
kAEPackedArray	Array items consist of data of the same size and same type, and are packed without regard for word boundaries.

Array items in OSA event arrays of type kAEDataArray, kAEPackedArray, or kAEHandleArray must be factored-that is, contained in a factored descriptor list. Before adding array items to a factored descriptor list, you should provide both a pointer to the data that is common to all array items and the size of that common data when you first call AECreateList to create a factored descriptor list. When you call AEPutArray to add the array data to such a descriptor list, the OSA Event Manager automatically isolates the common data you specified in the call to AECreateList.

When you call AEGetArray or AEPutArray, you specify a pointer of data type AEArrayDataPointer that points to a buffer containing the data for the array.

```
typedef AEArrayData *AEArrayDataPointer;
```

Routines for Responding to OSA events

This section describes the OSA Event Manager routines you can use to create and manage the OSA event dispatch tables, dispatch OSA events, extract information from OSA events, request user interaction, request more time to respond to OSA events, suspend and resume OSA event handling, delete descriptor records, deallocate memory for descriptor records, create and manage the coercion handler and special handler dispatch tables, and get information about the OSA Event Manager.

Creating and Managing the OSA Event Dispatch Tables

An OSA event dispatch table contains entries that specify the event class and event ID that refer to one or more OSA events, the address of the handler routine that handles those OSA events, and a reference constant. You can use the AEInstallEventHandler function to add entries to the OSA event dispatch table. This function sets up the initial mapping between the handlers in your application and the OSA events that they handle.

To get the address of a handler currently in the OSA event dispatch table, use the AEGetEventHandler function. If you need to remove any of your OSA event handlers after the mapping between handlers and OSA events is established, you can use the AERemoveEventHandler function.

Dispatching OSA Events

After receiving a PM message (and optionally determining whether it is an OSA event other than an OSA event that your application might support), your application typically calls the AEProcessOSAEvent function to determine the type of OSA event received and call the corresponding handler.

Getting Data or Descriptor Records Out of OSA Event Parameters a

The OSA Event Manager provides four functions that allow you to get data from OSA event parameters and attributes. The AEGetParamPtr and AEGetParamDesc functions get data from a specified OSA event parameter. The AEGetAttributePtr and AEGetAttributeDesc functions get data from a specified OSA event attribute.

Counting the Items in Descriptor Lists

The AECountItems function counts the number of descriptor records in any descriptor list, including an OSA event record.

Getting Items From Descriptor Lists

The OSA Event Manager provides three functions that allow you to get items from any descriptor list, including an OSA event record. The AEGetKeyPtr and AEGetKeyDesc functions give you access to the data in a descriptor list. The AEGetArray function gets data from an array contained in a descriptor list.

Getting Data and Keyword-Specified Descriptor Records Out of AE

The OSA Event Manager provides two functions, AEGetKeyPtr and AEGetKeyDesc, that allow you to get data and descriptor records out of an AE record or an OSA event record.

Requesting More Time to Respond to OSA events

The AEResetTimer function resets the timeout value for an OSA event to its starting value. A server application can call this function when it knows it cannot fulfill a client application's request (either by returning a result or by sending back a reply OSA event) before the client application is due to time out.

Suspending and Resuming OSA Event Handling

When your application calls AEProcessOSAEvent and one of your event handlers is invoked, the OSA Event Manager normally assumes that your application has finished handling the event when the event handler returns. At this point, the OSA Event Manager disposes of the event. However, some applications, such as multi-session servers or any applications that implement their own internal event queueing, may need to defer handling of the event.

The AESuspendTheCurrentEvent, AEResumeTheCurrentEvent, AESetTheCurrentEvent, and AEGetTheCurrentEvent functions described in this section allow you to suspend and resume OSA event handling, specify the OSA event to be handled, and identify an OSA event that is currently being handled.

Getting the Sizes and Descriptor Types of Descriptor Records

The OSA Event Manager provides four routines that allow you to get the sizes and descriptor types of descriptor records that are not part of an OSA event record. The AESizeOfNthItem function returns the size and descriptor type of a descriptor record in a descriptor list. The AESizeOfKeyDesc function returns the size and descriptor type of a keyword-specified descriptor record in an AE record. You can get the size and descriptor type of an OSA event parameter or OSA event attribute using the AESizeOfParam and AESizeOfAttribute functions.

Deleting Descriptor Records

The OSA Event Manager provides three functions that allow you to delete descriptor records. The AEDeleteItem, AEDeleteKeyDesc, and AEDeleteParam functions allow you to delete descriptor records from a descriptor list, an AE record, and an OSA event parameter, respectively.

Deallocating Memory for Descriptor Records

The AEDisposeDesc function deallocates the memory used by a descriptor record. Because all OSA event structures (except for keyword-specified descriptor records) are descriptor records, you can use AEDisposeDesc for any of them.

Coercing Descriptor Types

The OSA Event Manager provides two functions that allow you to coerce descriptor types. The AECoercePtr function takes a pointer to data and a desired descriptor type and attempts to coerce the data to a descriptor record of the desired descriptor type. The AECoerceDesc function attempts to coerce the data in an existing descriptor record to another descriptor type.

Creating and Managing the Coercion Handler Dispatch Tables

The OSA Event Manager provides three functions that allow you to create and manage the coercion handler dispatch tables. The AEInstallCoercionHandler function installs a coercion handler routine in either the application or system coercion dispatch table. The AEGetCoercionHandler function returns the handler for a specified descriptor type coercion. The AERemoveCoercionHandler function removes a coercion handler from either the application or system coercion table.

Creating and Managing the Special Handler Dispatch Tables

The OSA Event Manager provides three functions that allow you to create and manage the special handler dispatch tables. The AEInstallSpecialHandler function installs an entry for a special handler in either the application or system special handler dispatch table. The AEGetSpecialHandler function returns the handler for a specified special handler. The AERemoveSpecialHandler function removes a special handler from either the application or system special handler dispatch table.

You can also use the AEInstallSpecialHandler, AEGetSpecialHandler, and AERemoveSpecialHandler functions to install, get, and remove object callback functions-including system object callback functions, which cannot be installed with the AESetObjectCallbacks function. When calling any of these three functions, use one of the following constants as the value of the *functionClass* parameter to specify the object callback function:

Object callback function	Constant
Object-counting function	keyAECountProc
Object-comparison function	keyAECompareProc
Token disposal function	keyDiposeTokenProc
Error callback function	keyAEGetErrDescProc
Mark token function	keyAEMarkTokenProc
Object-marking function	keyAEMarkProc
Mark-adjusting function	keyAEAdjustMarksProc

You can also use the AERemoveSpecialHandler function to disable all the OSA Event Manager routines that support object specifier records. To do this, specify the constant keySelectProc in the *functionClass* parameter.

Getting Information About the OSA Event Manager

The AEManagerInfo routine allows you to get two kinds of information related to OSA events on the current computer: the number of processes currently recording OSA events and the version of the OSA Event Manager. If you decide to make your application recordable, this information may be useful when your application is responding to OSA events that it sends to itself.

Application-Defined Routines

For each OSA event your application supports, you must provide an OSA event handler. The AEProcessOSAEvent function calls one of your OSA event handlers when it processes an OSA event. An OSA event handler (MyEventHandler) should perform any action described by the OSA event, add parameters to the reply OSA event if appropriate, and return a result code.

You can also provide your own coercion handlers to coerce data to descriptor types other than those for which the OSA Event Manager provides coercion handling. The MyCoercePtr function accepts a pointer to data and returns a descriptor record, and the MyCoerceDesc function accepts a descriptor record and returns a descriptor record.

MyEventHandler

This function is an example of an OSA event handler.

Syntax

```
#define INCL_OSAEVENT

#include <os2.h>

OSErr MyEventHandler (OSAEvent *theOSAEvent, OSAEvent reply,
                      long handlerRefcon)
```

Parameters

theOSAEvent (OSAEvent *) - input

The OSA event to handle.

reply (OSAEvent) - input
The default reply OSA event provided by the OSA Event Manager.

handlerRefcon (long) - input
The reference constant stored in the OSA event dispatch table for the OSA event.

Returns

rc (OSErr) - returns
Return code.

Remarks

An OSA event handler should extract any parameters and attributes from the OSA event, perform the requested action, and add parameters to the reply OSA event if appropriate.

Your handler should always set its function result to noErr if it successfully handles the OSA event. If an error occurs, your handler should return either errAEEventNotHandled or some other nonzero result code. If the error occurs because your application cannot understand the event, return errAEEventNotHandled, in case a handler in the system special handler or system OSA event dispatch tables might be able to handle the event. If the error occurs because the event is impossible to handle as specified, return the result code returned by what ever function caused the failure, or whatever other result code is appropriate.

For example, suppose your application receives a Get Data event that requests the name of the current printer, and your application cannot handle such an event. In this situation, you should return errAEEventNotHandled in case another handler available to the OSA Event Manager can handle the event. This strategy allows users to take advantage of system capabilities from within your application via system handlers.

However, if your application cannot handle a Get Data event that requests the fifth paragraph in a document because the document contains only four paragraphs, you should return some other nonzero error, because further attempts to handle the event are pointless.

If your OSA event handler calls the AEResolve function and AEResolve calls an object accessor function in the system object accessor dispatch table, your OSA event handler may not recognize the descriptor type of the token returned by the function. In this case, your handler should return the result code errAEUnknownObjectType. When your handler returns this result code, the OSA Event Manager attempts to locate a system OSA event handler that can recognize the token.

MyCoercePtr

This function is an example of a coercion handler that accepts a pointer to data.

Syntax

```
#define INCL_OSAEVENT

#include <os2.h>

OSErr MyCoercePtr (DescType typeCode, void *dataPtr, Size dataSize,
                   DescType toType, long handlerRefcon, AEDesc *result)
```

Parameters

typeCode (DescType) - input
The descriptor type of the original data.

dataPtr (void *) - input
A pointer to the data to coerce.

dataSize (Size) - input
The length, in bytes, of the data to coerce.

toType (DescType) - input
The desired descriptor type for the resulting descriptor record.

handlerRefcon (long) - input
A reference constant that is stored in the coercion dispatch table entry for the handler and passed to the handler by the OSA Event Manager whenever the handler is called.

result (AEDesc *) - input

The resulting descriptor record.

Returns

rc (OSErr) - returns
Return code.

Remarks

Your coercion handler should coerce the data to the desired descriptor type and return the resulting data in the descriptor record specified by the result parameter. Your handler should return the *noErr* result code if your handler successfully performs the coercion, and a nonzero result code otherwise.

MyCoerceDesc

This function is an example of a coercion handler that accepts a descriptor record.

Syntax

```
#define INCL_OSAEVENT

#include <os2.h>

OSErr MyCoerceDesc (AEDesc *theAEDesc, DescType toType, long handlerRefcon,
                    AEDesc *result)
```

Parameters

theAEDesc (AEDesc *) - input
The descriptor record that contains the data to be coerced.

toType (DescType) - input
The desired descriptor type for the resulting descriptor record.

handlerRefcon (long) - input
A reference constant that is stored in the coercion dispatch table entry for the handler and passed to the handler by the OSA Event Manager whenever the handler is called.

result (AEDesc *) - input
The resulting descriptor record.

Returns

rc (OSErr) - returns
Return code.

Remarks

Your coercion handler should coerce the data in the descriptor record to the desired descriptor type and return the resulting data in the descriptor record specified by the *result* parameter. Your handler should return an appropriate result code.

Summary of Responding to OSA Events

The following table summarizes the methods used to respond to OSA events.

Function Name	Description
AEClearDesc	Initializes the specified descriptor record.
AECoerceDesc	Coerces the data in a descriptor record to another descriptor type.

AECoercePtr	Coerces the data to a desired descriptor type and, if successful, creates a descriptor record containing the newly coerced data.
AECountItems	Counts the number of descriptor records in any descriptor list.
AEDeleteItem	Deletes a descriptor record from a descriptor list. All subsequent descriptor records will then move up one place.
AEDeleteKeyDesc	Deletes a keyword-specific descriptor record from an AE record.
AEDeleteParam	Deletes an OSA event parameter
AEDisposeDesc	Deallocates the memory used by a descriptor record.
AEGetAppName	Returns the name of an application that sent an OSA event.
AEGetArray	Converts an OSA event array (an array created with the AEPutArray function and stored in a descriptor list) to the corresponding array and places the converted array in the specified buffer.
AEGetAttributeDesc	Returns the descriptor record for a specified OSA event attribute.
AEGetAttributePtr	Returns a pointer to a buffer that contains the data from a specified OSA event attribute.
AEGetCoercionHandler	Returns the handler for a specified descriptor type coercion.
AEGetDescData	Returns the descriptor type of the specified descriptor record and a copy of the data that it references.
AEGetEventHandler	Returns an entry from an OSA event dispatch table.
AEGetHWND	Returns the window handle from a specified PID of an application that has been registered through the AEInit function.
AEGetKeyDesc	Obtains the descriptor record for a keyword-specified descriptor record.
AEGetKeyPtr	Returns a pointer to a buffer that contains the data from a keyword-specified descriptor record.
AEGetNthDesc	Returns a copy of a descriptor record from any descriptor list.
AEGetNthPtr	Returns a pointer to a buffer that contains a copy of a descriptor record from any descriptor list.
AEGetParamDesc	Returns the descriptor record for a specified OSA event parameter.
AEGetParamPtr	Obtains a pointer to a buffer that contains the data from a specified OSA event parameter.
AEGetPID	Returns the process ID (PID) of the specified application that has issued an AEInit function.

AEGetSpecialHandler	Returns the specified special handler.
AEGetTheCurrentEvent	Returns the OSA event that is currently being handled.
AEInit	Registers a semantic-event aware application with the OSA Event Manager.
AEInstallCoercionHandler	Installs a coercion handler routine in the application's coercion handler dispatch table.
AEInstallEventHandler	Installs a coercion handler routine in the application's coercion handler dispatch table.
AEInstallSpecialHandler	Installs a special handler in the application's special handler dispatch table.
AELaunchApplication	Launches an application on a local machine.
AEManagerInfo	Obtains information about the version of the OSA Event Manager currently available or the number of processes that are currently recording OSA events.
AEProcessOSAEvent	Calls the appropriate handler for a specified OSA event.
AERemoveCoercionHandler	Removes a handler from the application's special handler dispatch table.
AERemoveEventHandler	Removes an entry form the application's OSA event dispatch table.
AERemoveSpecialHandler	Removes an entry form the application's special handler dispatch table.
AEResetTimer	Resets the timeout value for an OSA event to its starting value.
AEResumeTheCurrentEvent	Informs the OSA Event Manager that the application wants to resume the handling of a previously suspendedOSA event or that it has completed the handling of the OSA event.
AESetTheCurrentEvent	Specifies the OSA event to be handled.
AESizeOfAttribute	Returns the size and descriptor type of an OSA event attribute.
AESizeOfDescData	Returns the size of the descriptor data that is contained in the specified descriptor record.
AESizeOfKeyDesc	Returns the size and descriptor type of a keyword-specified descriptor record in an AE record.
AESizeOfNthItem	Returns the size and descriptor type of a descriptor record in a descriptor list.
AESizeOfParam	Returns the size and descriptor type of an OSA event parameter.
AESuspendTheCurrentEvent	Suspends the processing of the OSA event that is currently being handled.
AETerminate	Cleans up internal data structures when an application ends.

Creating and Sending OSA Events

This chapter describes how your application can use the OSA Event Manager to create and send OSA events. If you want to factor your application for recording, or if you want your application to send OSA events directly to other applications, you can use OSA Event Manager routines to create and send OSA events.

Before you read this chapter, you should be familiar with Interapplication Communication, OSA Events, and Responding to OSA Events.

This chapter provides the basic information you need to create and send OSA events from your application. To send core and functional-area OSA events, your application must also be able to create object specifier records. For information about object specifier records, see Resolving and Creating Object Specifier Records.

The first section in this chapter, Creating an OSA Event, describes how to:

- Create an OSA event
- Add parameters to an OSA event
- Specify optional OSA event parameters
- Specify a target address

The section Sending an OSA Event describes how to:

- Send an OSA event
- Deal with time-outs

Creating an OSA Event

You create an OSA event using the AECreateOSAEvent function. You supply parameters that specify the event class and event ID, the target address, the return ID, the transaction ID, and a buffer for the returned OSA event. The AECreateOSAEvent function creates and returns, in the buffer you specify, an OSA event with the attributes set as your application requested. You should not directly manipulate the contents of the OSA event; rather, use OSA Event Manager functions to add additional attributes or parameters to it.

The example that follows creates an imaginary Multiply event using the AECreateOSAEvent function:

```
myErr = AECreateOSAEvent (kArithmeticClass, kMultEventID, &targetAddress,
                          kAutoGenerateReturnID, kAnyTransactionID,
                          &theOSAEvent);
```

The event class, specified by the kArithmeticClass constant, identifies this event as belonging to a class of OSA events for arithmetic operations. The event ID specifies the particular OSA event within the class-in this case, an OSA event that performs multiplication.

You specify the target of the OSA event in the *target* parameter to AECreateOSAEvent. The target address can identify an application on the local computer. You can specify the address using a process ID (PID).

In the *returnID* parameter, you specify the return ID of the OSA event, which associates this OSA event with the server's reply. The AECreateOSAEvent function assigns the specified return ID value to the keyReturnIDAttr attribute of the OSA event. If a server returns a standard reply OSA event (that is, an event of event class **aevt** and event ID **ansr**) in response to this event, the OSA Event Manager assigns the reply event the same return ID. When you receive a reply OSA event, you can check the keyReturnIDAttr attribute to determine which outstanding OSA event the reply is responding to. You can use the kAutoGenerateReturnID constant to request that the OSA Event Manager generate a return ID that is unique to this session for the OSA event. Otherwise, you are responsible for making it unique.

The *transactionID* parameter specifies the transaction ID attribute of the OSA event. A *transaction* is a sequence of OSA events that are sent back and forth between the client and server applications, beginning with the client's initial request for a service. All OSA events that are part of one transaction must have the same transaction ID.

You can use a transaction ID to indicate that an OSA event is one of a sequence of OSA events related to a single transaction. The kAnyTransactionID constant indicates that the OSA event is not part of a transaction.

The AECreateOSAEvent function creates an OSA event with only the specified attributes and no parameters. To add parameters or additional attributes, you can use other OSA Event Manager functions.

Adding Parameters to an OSA Event

You can use the AEPutParamPtr or AEPutParamDesc function to add parameters to an OSA event. When you use either of these functions, the OSA Event Manager adds the specified parameter to the OSA event.

Use the AEPutParamPtr function when you want to add data specified in a buffer as the parameter of an OSA event. You specify the OSA event, the keyword of the parameter to add, the descriptor type, a buffer that contains the data, and the size of this buffer as parameters to the AEPutParamPtr function. The AEPutParamPtr function adds the data to the OSA event as a parameter with the specified keyword.

For example, this code adds a parameter to the Multiply event using the AEPutParamPtr function:

```
#define   keyOperand1 '1NPO'

long     number1;
OSAEvent theOSAEvent;
OSErr     myErr;

number1 = 10;
myErr = AEPutParamPtr (&theOSAEvent, keyOperand1, typeLongInteger, &number1
                       sizeof(number1));
```

In this example, the OSA Event Manager adds the parameter containing the first number to the specified OSA event.

Use the AEPutParamDesc function to add a descriptor record to an OSA event. The descriptor record you specify must already exist. To create or get a descriptor record, you can use the AECreateDesc, AEDuplicateDesc, and other OSA Event Manager functions that return a descriptor record.

When you create a descriptor record using the AECreateDesc function, you specify the descriptor type, a buffer that contains the data, and the size of this buffer as parameters. The AECreateDesc function returns the descriptor record that describes the data.

This example creates a descriptor record for the second parameter of the Multiply event:

```
long     number2;
AEDesc   multParam2Desc;
OSErr   myErr;

number2 = 8;
myErr = AECreateDesc(typeLongInteger, &number2, sizeof(number2),
                     &multParam2Desc);
```

In this example, the AECreateDesc function creates a descriptor record with the typeLongInteger descriptor type and the data identified in the number2 variable.

Once you have created a descriptor record, you can use AEPutParamDesc to add the data to an OSA event parameter. You specify the OSA event to add the parameter to, the keyword of the parameter, and the descriptor record of the parameter as parameters to the AEPutParamDesc function.

This example adds a second parameter to the Multiply event using the AEPutParamDesc function:

```
#define keyOperand2   '2NPO'

myErr = AEPutParamDesc (&theOSAEvent, keyOperand2, &multParam2Desc);
```

This example adds the keyOperand2 keyword and the descriptor record created in the previous example as the second parameter to the

specified OSA event.

Whatever method you use to create a descriptor record, you can add it to an OSA event parameter by using the AEPutParamDesc function.

After adding parameters to an OSA event, you can send the OSA event using the AESend function. See Sending an OSA Event for information about using this function.

Specifying Optional Parameters for an OSA Event

The parameters for a given OSA event are listed in the *OSA Event Registry: Standard Suite*s as either required or optional. Your application does not usually have to include OSA event parameters that are listed as optional; the target application uses default values for parameters that are listed as optional if your application does not provide them. The *OSA Event Registry: Standard Suite*s defines the default value a target application should use for each optional parameter of a specific OSA event.

The guidelines listed in the *OSA Event Registry: Standard Suite*s for which parameters should be considered optional and which should be considered required are not enforced by the OSA Event Manager. Instead, the source application indicates which OSA event parameters it considers optional by listing the keywords for those parameters in the keyOptionalKeywordAttr attribute.

The keyOptionalKeywordAttr attribute does not contain the optional parameters; it simply lists the keywords of any parameters for the OSA event that the source application wants to identify as optional. Although the source application is responsible for providing this information in the keyOptionalKeywordAttr attribute of an OSA event, it is not required to provide this attribute.

If a keyword for an OSA event parameter is not included in the keyOptionalKeywordAttr attribute, the source application expects the target application to understand the OSA event parameter identified by that keyword. If a target application cannot understand the parameter, it should return the result code errAEParamMissed and should not attempt to handle the event.

If a keyword for an OSA event parameter is included in the keyOptionalKeywordAttr attribute, the source application does not require the target application to understand the OSA event parameter identified by that keyword. If the target application cannot understand a parameter whose keyword is included in the keyOptionalKeywordAttr attribute, it should ignore that parameter and attempt to handle the OSA event as it normally does.

A source application can choose not to list the keyword for an OSA event parameter in the keyOptionalKeywordAttr attribute even if that parameter is listed in the *OSA Event Registry: Standard Suites* as an optional parameter. This has the effect of forcing the target application to treat the parameter as required for a particular OSA event. If the target application supports the parameter, it should handle the OSA event as the client application expects. If the target application does not support the parameter and calls an application-defined routine such as MyGotRequiredParams to check whether it has received all the required parameters, it finds that there is another parameter that the client application considered required, and should return the result code errAEParamMissed.

If a source application wants a target application to attempt to handle an OSA event regardless of whether the target application supports a particular OSA event parameter included in that OSA event, the source application should list the keyword for that parameter in the keyOptionalKeywordAttr attribute.

It is up to the source application to decide whether to list a parameter that is described as optional in the *OSA Event Registry: Standard Suites* in the keyOptionalKeywordAttr attribute of an OSA event. For example, suppose a source application has extended the definition of the Print event to include an optional keyColorOrGrayscale parameter that specifies printing in color or gray scale rather than black and white. The source application might decide whether or not to list the keyword keyColorOrGrayscale in the keyOptionalKeywordAttr attribute according to the characteristics of the print request. If the source application requires the target application to print a document in color, the source application could choose not to add the keyword keyColorOrGrayscale to the keyOptionalKeywordAttr attribute; in this case, only target applications that supported the keyColorOrGrayscale parameter would attempt to handle the event. If the source application does not require the document printed in color, it could choose to add the keyword keyColorOrGrayscale to the keyOptionalKeywordAttr attribute; in this case, the target application will attempt to handle the event regardless of whether it supports the keyColorOrGrayscale parameter.

Your application can add optional parameters to an OSA event the same way it adds required parameters, using the AECreateDesc, AEPutParamPtr, and AEPutParamDesc functions as described in the previous section, Adding Parameters to an OSA Event. If your application chooses to provide the keyOptionalKeywordAttr attribute for an OSA event, it should first create a descriptor list that specifies the keywords of the optional parameters, then add it to the OSA event as a keyOptionalKeywordAttr attribute.

The following code fragment shows an application-defined routine, MyCreateOptionalKeyword, that creates the keyOptionalKeywordAttr attribute for the Save event.

```
OSErr MyCreateOptionalKeyword (OSAEvent *SaveEvent)
{
  AEDescList optionalList;
  AEKeyword myOptKeyword1, keyAEFileType;
  AEKeyword myOptKeyword2, keyAEFile;
  OSErr     myErr;
  OSErr     ignoreErr;
```

```
    myOptKeyword1 = keyAEFileType;
    /* create an empty descriptor list */
    myErr = AECreateList(NULL, 0, FALSE, &optionalList);
    if (myErr == noErr) {
        /* add the keyword of the first optional parameter */
        myErr = AEPutPtr(&optionalList, 1, typeKeyword, &myOptKeyword1,
                         sizeof(myOptKeyword1));
        if (myErr == noErr) {
            /* add the keyword of the next optional parameter */
            myOptKeyword2 = keyAEFile;
            myErr = AEPutPtr(&optionalList, 2, typeKeyword, &myOptKeyword2,
                             sizeof(myOptKeyword2));
        }
        if (myErr == noErr)
            /* create the keyOptionalKeywordAttr attribute and add it */
            /* to the Save event */
            myErr = AEPutAttributeDesc(SaveEvent, keyOptionalKeywordAttr,
                                       &optionalList);
    }
    ignoreErr = AEDisposeDesc(&optionalList);
    return (myErr);
}
```

The MyCreateOptionalKeyword function shown in the previous code fragment adds to a descriptor list the keyword of each parameter that the source application considers optional. Each keyword is added as a descriptor record with the descriptor type typeKeyword. The function specifies that the target application can handle the Save event without supporting parameters identified by the keywords keyAEFileType and keyAEFile. After adding these keywords to the descriptor list, the function creates the keyOptionalKeywordAttr attribute using the AEPutAttributeDesc function.

Typically, a target application does not examine the keyOptionalKeywordAttr attribute directly. Instead, a target application that supports a parameter listed as optional in the *OSA Event Registry: Standard Suite*s attempts to extract it from the OSA event (using AEGetParamDesc, for example). If it cannot extract the parameter, the target application uses the default value, if any, listed in the OSA Event Registry. A target application can use the keyMissedKeywordAttr attribute to return the first required parameter (that is, considered required by the source application), if any, that it did not retrieve from the OSA event. The keyMissedKeywordAttr attribute does not return any parameters whose keywords are listed in the keyOptionalKeywordAttr attribute of the OSA event.

Specifying a Target Address

When you create an OSA event, you must specify the address of the target. The *target addr*ess identifies the particular application or process to which you want to send the OSA event. You can send OSA events to applications on the local computer.

The descriptor type that identifies the method of addressing an OSA event is:

Descriptor Type	**Description**
typePID	Process ID

If your application sends an OSA event to itself, it should address the OSA event using the PID of kCurrentProcess. This is the fastest way for your application to send an OSA event to itself. For more information, see Addressing an OSA Event for Direct Dispatching.

Creating an Address Descriptor Record

You specify the address using an address descriptor record (a descriptor record of data type AEAddressDesc). You must create a descriptor record of this type and then add it to the OSA event using the AECreateOSAEvent function.

You can use the AECreateDesc function to create an address descriptor record. The following code fragment shows an example of creating an address.

```
OSErr MySetTargetAddresses(AEAddressDesc targetAddress, DescType thePSID)
{
  DescType   thePID;
  OSErr     myErr;

  myErr = AECreateDesc(thePSID, &thePID, sizeof(thePID), &targetAddress);

  /* add your own error checking */
}
```

To create an address descriptor record, specify the following as parameters to AECreateDesc: the descriptor type for the address, a pointer to the buffer containing the address, and the size of the buffer. The AECreateDesc function returns an address descriptor record with the specified characteristics.

After creating an address, you can specify it as a parameter to the AECreateOSAEvent function. See Creating an OSA Event for an example using the AECreateOSAEvent function.

When you specify an address to the AECreateOSAEvent function, the OSA Event Manager stores the address in the keyAddressAttr attribute of the OSA event.

Addressing an OSA Event for Direct Dispatching

As described in Recording OSA Events, a recordable application must send itself OSA events in response to user actions. Your application can send itself OSA events by using an address descriptor record of descriptor type typePID with the PID field set to kCurrentProcess. The OSA Event Manager processes such OSA events immediately, executing the appropriate OSA event handler directly without going through the normal event-processing sequence. Your application's performance is not decreased when sending OSA events to itself.

OSA events your application sends to itself this way do not appear in your application's PM message queue. This not only speeds up delivery of the event but also avoids situations in which an OSA event sent in response to a user action arrives in the event queue after some other event that really occurred later than the user action. For example, suppose a user chooses **Cut** from the **Edit** menu and then clicks in another window. If the Cut event arrives in the queue after the window activate event, a selection in the wrong window might be cut.

Your application can send events to itself using other forms of addressing, such as the true PID returned by AEGetPID. Because direct dispatching avoids event sequence problems, applications should generally send events to themselves by using an address descriptor record of descriptor type typePID with the kCurrentProcess constant rather than using a true PID or an application signature.

Important: When OSA event recording has been turned on, the OSA Event Manager records every event that your application sends to itself unless you specify the kAEDontRecord flag in the *sendMode* parameter of the AESend function.

Sending an OSA Event

To send an OSA event, you first create an OSA event, add parameters and attributes to it, and then use the AESend function to send it.

When you send an OSA event, you specify various options to indicate how the server should handle the OSA event. You request a user interaction level from the server whether your application is willing to wait for a reply OSA event, whether reconnection is allowed, and whether your application wants a return receipt for the OSA event.

You specify these options by setting flags in the *sendMode* parameter for AESend. Here are the constants that represent these flags:

kAENoReply	Sender does not want a reply
kAEQueueReply	Sender wants a reply but will not wait
kAEWaitReply	Sender wants a reply and will wait
kAENeverInteract	Server should not interact with user
kAECanInteract	Server may try to interact with user
kAEAlwaysInteract	Server should always interact with user where appropriate
kAECanSwitchLayer	Interaction may switch layer
kAEDontReconnect	Reserved for future use.
kAEWantReceipt	Sender wants a receipt of message
kAEDontRecord	Do not record this event
kAEDontExecute	Record but do not send the event

If you want your application to receive a reply OSA event, specify the kAEQueueReply or kAEWaitReply flag. If you want your application to

receive the reply OSA event in its event queue, use kAEQueueReply. If you want your application to receive the reply OSA event in the *reply* parameter for AESend and you are willing to block the current thread while it is waiting for the reply, use kAEWaitReply. If you do not want your application to receive a reply OSA event and your application does not need to wait for the server to handle the OSA event, specify kAENoReply.

Note: Before the OSA Event Manager sends a reply event back to the client application, the keyAddressAttr attribute contains the address of the client application. After the client receives the reply event, the keyAddressAttr attribute contains the address of the server application.

If you specify the kAEWaitReply flag, your application must call AESend from a separate thread. This allows the main thread to respond to any incoming events sent to the PM message queue.

You use one of the three flags-kAENeverInteract, kAECanInteract, and kAEAlwaysInteract-to specify whether the server should interact with the user when handling the OSA event. Specify kAENeverInteract if the server should not interact with the user when handling the OSA event. You might specify this constant if you do not want the user to be interrupted while the server is handling the OSA event.

Use the kAECanInteract flag if the server should interact with the user when the user needs to supply information to the server. Use the kAEAlwaysInteract flag if the server should interact with the user whenever the server normally asks a user to confirm a decision or interact in any other way, even if no additional information is needed from the user. Note that it is the responsibility of the server and client applications to agree on how to interpret the kAEAlwaysInteract flag.

If the client application does not set any one of the user interaction flags, the OSA Event Manager sets a default. The OSA Event Manager sets the kAECanInteract flag as the default.

Specify the kAEWantReceipt flag if your application wants notification that the server application has accepted the OSA event. If you specify this flag, your application receives a return receipt as a high-level event.

If you specify the kAEWantReceipt flag and the server application does not accept the OSA event within the time specified by the *timeOutInTicks* parameter to AESend, the AESend function returns a time-out error. Note that AESend also returns a time-out error if your application sets the kAEWaitReply flag and does not receive the reply OSA event within the time specified by the *timeOutInTicks* parameter.

Specify the kAEDontRecord flag if your application is sending an OSA event to itself that you do not want to be recorded. When OSA event recording has been turned on, every event that your application sends to itself will be automatically recorded by the OSA Event Manager except those sent with the kAEDontRecord flag set.

Specify the kAEDontExecute flag if your application is sending an OSA event to itself for recording purposes only-that is, if you want the OSA Event Manager to send a copy of the event to the recording process but you do not want your application actually to receive the event. (For more information about when to use the kAEDontExecute flag, see Recording OSA Events.)

The following code fragment illustrates how to send a Multiply event (an imaginary OSA event for multiplying two long integers). It first creates an OSA event, adds parameters containing the numbers to multiply, then sends it, specifying various options. It also illustrates how to handle the reply OSA event that contains the result.

Note: If you want to send OSA events, your application must call AEInit.

```
OSErr MySendMultiplyEvent (AEAddressDesc serverAddress, long firstOperand,
                           long secondOperand, long replyResult)
{

#define kArithmeticClass 0x48545241 /* 'ARTH' -event class for arithmetic *
                                     /*         OSA event. */
#define kMultiplyEventID 0x544C554D /* 'MULT' -event ID for Multiply event*
#define keyMultOperand1   0x314E504F /* 'OPN1' -keyword for first parameter
#define keyMultOperand2   0x324E504F /* 'OPN2' -keyword for second paramete

  OSAEvent theOSAEvent;
  OSAEvent reply;
  DescType returnedType;
  long     actualSize, replyResultLongInt;
  OSErr     myErr;
  OSErr     ignoreErr;
  ULONG     STRSIZE=256;
  char     errStr[256];
  long     errNumber;

  myErr = AECreateOSAEvent(kArithmeticClass, kMultiplyEventID,
                           &serverAddress, kAutoGenerateReturnID,
                           kAnyTransactionID, &theOSAEvent);
```

```
    /* add the first operand */
    if (myErr == noErr)
        myErr = AEPutParamPtr(&theOSAEvent, keyMultOperand1, typeLongInteger,
                              &firstOperand, sizeof(firstOperand));
    /*add the second operand with the proper keyword*/
    if (myErr == noErr)
        myErr = AEPutParamPtr(&theOSAEvent, keyMultOperand2, typeLongInteger,
                              &secondOperand, sizeof(secondOperand));
    if (myErr == noErr)
        myErr = AESend(&theOSAEvent, &reply, kAEWaitReply + kAENeverInteract,
                       kAENormalPriority, 120, NULL, NULL);
    if (myErr == noErr) {
        /* OSA event successfully sent */
        /* Check whether it was successfully handled-- */
        /* get result code returned by the server's handler */
        myErr = AEGetParamPtr(&reply, keyErrorNumber, typeLongInteger,
                              &returnedType, &errNumber, sizeof(errNumber),
                              &actualSize);
        if ((myErr == errAEDescNotFound) || (errNumber == noErr))
          /* if keyErrorNumber does not exist or server returned noErr */
          /* then the OSA event was successfully handled--the reply OSA */
          /* event contains the result in the direct parameter */
          myErr = AEGetParamPtr(&reply, keyDirectObject, typeLongInteger,
                                &returnedType, &replyResultLongInt,
                                sizeof(replyResultLongInt), &actualSize);
        else {
          /* server returned an error, so get error string */
          myErr = AEGetParamPtr(&reply, keyErrorString, typeChar, &returnedTy
                                &errStr[1], sizeof(errStr)-1,
                                &actualSize);
          if (myErr == noErr) {
              if (actualSize > 255)
                  actualSize = 255;
              errStr[0] = actualSize;
              MyDisplayError(errStr);
            }
        }
        ignoreErr = AEDisposeDesc(&reply);
    }
    else {
        /* the OSA event was not successfully dispatched, */
        /* the request timed out, the user canceled, or other error */
    }
    ignoreErr = AEDisposeDesc(&theOSAEvent);
    return (myErr);
}
```

The code in the previous code fragment first creates an OSA event with kArithmeticClass as the event class and kMultiplyEventID as the event ID. It also specifies the server of the OSA event. See Specifying a Target Address for information on specifying a target address and Creating an OSA Event for more information on creating an OSA event.

The Multiply event shown in the previous code fragment contains two parameters, each specifying a number to multiply. See Adding Parameters to an OSA Event for examples of how to specify the parameters for the AEPutParamPtr function.

After adding the parameters to the event, the code uses AESend to send the event. The *theOSAEvent* parameter of the AESend function specifies the OSA event to send-in this example, the Multiply event. The *reply* parameter specifies the reply OSA event.

This example specifies kAEWaitReply in the *sendMode* parameter, indicating that the client is willing to wait for the specified time-out value

(120 ticks, or 2 seconds). The kAENeverInteract flag indicates that the server should not interact with the user when processing the OSA event. The s*endPriority* parameter specifies that the Multiply event is to be sent using normal priority. You can specify the kAEHighPriority flag, but this is not usually recommended.

If you specify kAEWaitReply, a noErr result code from AESend indicates that the OSA event was sent successfully, not that the server has completed the requested action successfully. Therefore, you should find out whether a result code was returned from the handler by checking the reply OSA event for the existence of either the keyErrorNumber or keyErrorString parameter. If the keyErrorNumber parameter does not exist or contains the noErr result code, you can use AEGetParamPtr to get the parameter you are interested in from the reply OSA event.

The MySendMultiplyEvent function in the previous code fragment checks the function result of AESend. If it is noErr, MySendMultiplyEvent checks the keyErrorNumber parameter of the reply OSA event to determine whether the server successfully handled the OSA event. If this parameter exists and indicates that an error occurred, MySendMultiplyEvent gets the error string out of the keyErrorString parameter; otherwise, the server performed the request, and the reply OSA event contains the answer to the multiplication request.

When you have finished using the OSA event specified in the AESend function and no longer need the reply OSA event, you must dispose of both the original event and the reply by calling the AEDisposeDesc function.

Important: If your application sends OSA events to itself using a typePID address descriptor record set to kCurrentProcess, the OSA Event Manager jumps directly to the appropriate OSA event handler without going through the normal event-processing sequence. For this reason, your application will not appear to run more slowly when it sends OSA events to itself. For more information, see Addressing an OSA Event for Direct Dispatching.

If the kAEHighPriority flag is specified in the s*endPriority* parameter of the AESend method, the OSA Event Manager spawns a thread and uses the WinSendMessage function to deliver the event to the target process. This has the effect of by-passing the target process's message queue.

Dealing with Time-Outs

When your application calls AESend and chooses to wait for the server application to handle the OSA event, it can also specify the maximum amount of time it is willing to wait for a response. You can specify a time-out value in the *timeOutInTicks* parameter to AESend. You can either specify a particular length of time, in ticks, that your application is willing to wait, or you can specify the kNoTimeOut constant or the kAEDefaultTimeout constant.

Use the kNoTimeOut constant to indicate that your application is willing to wait forever for a response from the server. You should use this value only if you are sure that the server will respond in a reasonable amount of time. You should also implement a method of checking whether the user wants to cancel.

Use the kAEDefaultTimeout constant if you want the OSA Event Manager to use a default time-out value. The OSA Event Manager uses a time-out value of about one minute if you specify this constant.

If you set the kAEWaitReply flag and the server does not have a handler for the OSA event, the server immediately returns the errAEEventNotHandled result code. If the server does not respond within the length of time specified by the time-out value, AESend returns the errAETimeout result code and a reply OSA event that contains no data. This result code does not necessarily mean that the server failed to perform the requested action; it means only that the server did not complete processing within the specified time. The server might still be processing the OSA event, and it might still send a reply.

If the server finishes processing the OSA event sometime after the time specified in the keyTimeoutAttr attribute has expired, it returns a reply OSA event to AEProcessOSAEvent. The OSA Event Manager then adds the actual data to the reply. Thus, your application can continue to check the reply OSA event to see if the server has responded, even after the time expires. If the server has not yet sent the reply when the client attempts to extract data from the reply OSA event, the OSA Event Manager functions return the errAEReplyNotArrived result code. After the reply OSA event returns from the server, the client can extract the data in the reply.

Additionally, the server can examine the keyTimeoutAttr attribute of the OSA event to determine the time-out value specified by the client. You can use the value of this attribute as a rough estimate of how much time your handler has to respond. You can assume that your handler has less time to respond than this time-out value, because transmitting the OSA event uses some of the available time, as does transmitting the reply OSA event back to the client, and the event may have been in the queue for a while already.

If you set the kAENoReply or kAEQueueReply flag, the OSA Event Manager ignores any time-out value you specify, because your application is not waiting for the reply. An attempt by the server to examine the keyTimeoutAttr attribute in this situation generates the error errAEDescNotFound.

If your handler needs more time than is specified in the keyTimeoutAttr attribute, you can reset the timer by using the AEResetTimer function. This function resets the time-out value of an OSA event to its starting value.

Routines for Creating and Sending OSA Events

This section describes the basic OSA Event Manager routines that your application can use to create OSA events, create and duplicate

descriptor records, create and add items to descriptor lists and AE records, add parameters and attributes to OSA events, and send OSA events. The section Routines for Responding to OSA events describes other OSA Event Manager routines used for both responding to and creating OSA events.

Creating OSA Events

The AECreateOSAEvent function allows you to create an OSA event.

Creating and Duplicating Descriptor Records

The AECreateDesc function converts data into a descriptor record, and the AEDuplicateDesc function makes a copy of a descriptor record.

Creating Descriptor Lists and AE Records

The AECreateList function allows you to create an empty descriptor list or AE record.

Adding Items to Descriptor Lists

The OSA Event Manager provides three routines that allow you to add descriptor records to any descriptor list, including an OSA event record. The AEPutPtr function converts data specified in a buffer to a descriptor record and adds the descriptor record to a descriptor list. The AEPutDesc function adds a descriptor record to a descriptor list. The AEPutArray function puts the data for an OSA event array into a descriptor list.

Adding Data and Descriptor Records to AE Records

The OSA Event Manager provides two routines that allow you to add data and descriptor records to AE records. The AEPutKeyPtr function takes a pointer to data, a descriptor type, and a keyword and converts them into a keyword-specified descriptor record that it adds to an AE record. The AEPutKeyDesc function takes a descriptor record and a keyword and converts them into a keyword-specified descriptor record that it adds to an AE record.

Adding Parameters and Attributes to OSA Events

The OSA Event Manager provides four functions that allow you to add OSA event parameters and attributes to an OSA event. The AEPutParamPtr and AEPutParamDesc functions add parameters to a specified OSA event. The AEPutAttributePtr and AEPutAttributeDesc functions add attributes to a specified OSA event.

Sending OSA Events

The AESend function allows you to send an OSA event that you have previously created with the AECreateOSAEvent function.

Summary of Creating and Sending OSA Events

The following table summarizes the methods used to create and send OSA events.

Function Name	Description
AECreateDesc	Converts data into a descriptor record.
AECreateList	Creates an empty descriptor list of AE records.
AECreateOSAEvent	Creates an OSA event with several important attributes but no parameters.
AEDuplicateDesc	Makes a copy of a descriptor record.
AEPutArray	Puts the data for an OSA event array into any descriptor list.
AEPutAttributeDesc	Adds a descriptor record and a keyword to an OSA event as an attribute.
AEPutAttributePtr	Adds a pointer to data, a descriptor type, and a keyword to an OSA event as an attribute.
AEPutDesc	Adds a descriptor to any descriptor list.
AEPutKeyDesc	Adds a descriptor record and a keyword to an AE record as a keyword-specified descriptor record.
AEPutKeyPtr	Gets a pointer to a buffer that contains the data from a keyword-specified descriptor record.
AEPutParamDesc	Adds a descriptor record and a keyword to an OSA event as an OSA event parameter.
AEPutParamPtr	Adds a pointer to data, a descriptor type, and a keyword to an OSA event as an OSA event parameter.
AEPutPtr	Adds data specified in a buffer to any descriptor list as a descriptor record.
AESend	Sends an OSA event.

Resolving and Creating Object Specifier Records

This chapter describes how your application can use the OSA Event Manager and application-defined functions to resolve object specifier records. Your application must be able to resolve object specifier records to respond to core and functional-area OSA events defined in the *OSA Event Registry: Standard Suites* .

For example, after receiving a Get Data event that requests a table in a document, your application can use the OSA Event Manager and application-defined functions to parse the object specifier record in the *direct* parameter, locate the requested table, and send a reply OSA event containing the table's data back to the application that requested it.

This chapter also describes how your application can use the OSA Event Manager to create object specifier records. If you want to factor your application for OSA event recording, or if you want to send OSA events directly to other applications, you need to know how to create object specifier records.

To use this chapter, you should be familiar with OSA Events and Responding to OSA Events. The section Working with Object Specifier

Records provides a general introduction to the subject.

If you plan to create object specifier records, you should also be familiar with Creating and Sending OSA Events. If you are factoring your application, you should read Recording OSA Events before you write code for resolving or creating object specifier records.

This chapter begins with an overview of the way your application works with the OSA Event Manager to resolve object specifier records. It then describes:

- How the data in an object specifier record is organized
- How to install entries in the object accessor tables
- How to write object accessor and object callback functions
- How to create an object specifier record

Resolving Object Specifier Records

If an OSA event parameter consists of an object specifier record, your handler for the OSA event should *resolve* the object specifier record: that is, locate the OSA event objects it describes. The first step is to call the AEResolve function with the object specifier record as a parameter.

The AEResolve function performs tasks that are required to resolve any object specifier record, such as parsing its contents, keeping track of the results of tests, and handling memory management. When necessary, AEResolve calls application-defined functions to perform tasks that are unique to the application, such as locating a specific OSA event object in the application's data structures or counting the number of OSA event objects in a container.

Note: Object specifier records are only valid while the OSA event that contains them is being handled. For example, if an application receives an OSA event asking it to cut row 5 of a table, what was row 6 then becomes row 5, and the original object specifier record that referred to row 5 no longer refers to the same row.

The AEResolve function can call two kinds of application-defined functions. *Object accessor functions* locate OSA event objects. *Object callback functions* perform other tasks that only an application can perform, such as counting, comparing, or marking OSA event objects. This section provides an overview of the way AEResolve calls object accessor and object callback functions when it resolves object specifier records.

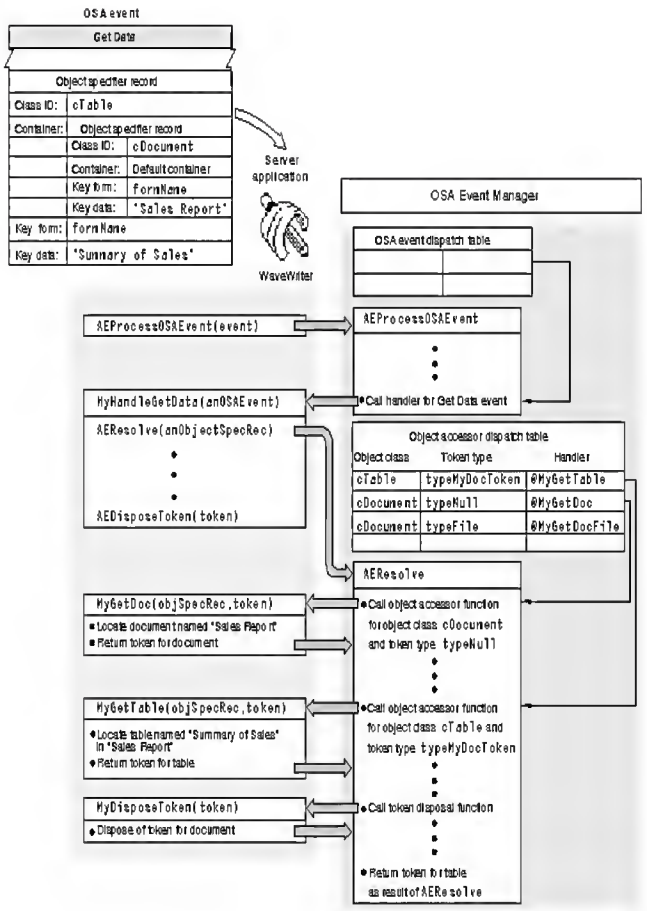
Each time AEResolve calls one of your application's object accessor functions successfully, the object accessor function should return a special descriptor record created by your application, called a *token* , that identifies either an element in a specified container or a property of a specified OSA event object. The OSA Event Manager examines the token's descriptor type but does nothing with the token's data. When it needs to refer to the object the token identifies, the OSA Event Manager simply passes the token back to your application.

Each object accessor function provided by your application should either find elements of a given object class in a container identified by a token of a given descriptor type or find properties of an OSA event object identified by a token of a specified descriptor type. The OSA Event Manager uses the object class ID and the descriptor type of the token that identifies the object's container to determine which object accessor function to call.

It is up to you to decide how many object accessor functions you need to write for your application. You can write one object accessor function that locates OSA event objects of several different object classes, or you can write separate object accessor functions for certain object classes. Similarly, you may want to use only one descriptor type for all the tokens returned by your object accessor functions, or you may want to use several descriptor types. The way you define your tokens depends on the needs of your application.

You can use the AEInstallObjectAccessor function to create an *object accessor dispatch table* that the OSA Event Manager uses to map requests for OSA event objects to the appropriate object accessor function in your application. The OSA Event Manager uses the object class of each requested object and the descriptor type of the token that identifies the object's container to determine which object accessor function to call. Depending on the container hierarchy for a given object specifier record and the way your application defines its object accessor functions, the OSA Event Manager may need to call a series of object accessor functions to resolve the nested object specifier records that describe an OSA event object's container. For information about creating and using the object accessor dispatch table, see Installing Entries in the Object Accessor Dispatch Tables.

The following figure illustrates the major steps involved in resolving an object specifier record. The WaveWriter application shown in the following figure receives a Get Data event whose direct parameter is an object specifier record for a table named "Summary of Sales" in a document named "Sales Report." The WaveWriter application's handler for the Get Data event calls the AEResolve function with the object specifier record as a parameter. The AEResolve function begins to parse the object specifier record. The first object accessor function that AEResolve calls is usually the function that can identify the OSA event object in the application's *default container* -the outermost container in the container hierarchy. In the following figure, the object specifier record for the document "Sales Report" specifies the default container, so the OSA Event Manager calls the object accessor function in the WaveWriter application that can locate a document in a container identified by a descriptor record of descriptor type typeNull.



After locating the document named "Sales Report," the WaveWriter application returns a token to the OSA Event Manager-that is, a descriptor record that WaveWriter uses to identify the document. The OSA Event Manager examines the descriptor type of the token but does not need to know anything about the token's data to continue parsing the object specifier record. Next, the OSA Event Manager calls the object accessor function that can identify a table in a container identified by a token of descriptor type typeMyDocToken. When the OSA Event Manager calls this object accessor function, it uses the token that describes the document to identify the table's container. After the WaveWriter application has located the table named "Summary of Sales" in the document named "Sales Report," it returns a token describing that table to the OSA Event Manager.

After your application has successfully located an OSA event object, the OSA Event Manager disposes of all previous tokens returned during resolution of the object specifier record for the object. The OSA Event Manager disposes of tokens by calling either the AEDisposeDesc function or your application's *token disposal function* , if you have provided one, which is an object callback function that disposes of a token. In the previous figure, the AEResolve function calls the WaveWriter application's token disposal function to dispose of the token for the document after AEResolve receives the token for the table. After the WaveWriter application has disposed of the token for the document, the AEResolve function returns the result of the resolution-that is, the token for the requested table-to the handler in the WaveWriter application that originally called AEResolve.

The OSA Event Manager can complete the cycle of parsing the object specifier record and calling the appropriate object accessor function to obtain a token as many times as necessary to identify every container in the container hierarchy and finish resolving an object specifier record, including disposing of the tokens for the containers. However, one token will always be left over-the token that identifies the requested OSA event object. After AEResolve returns this final token and your application performs the action requested by the OSA event, it is up to your application to dispose of the token. Your application can do so by calling the AEDisposeToken function, which in turn calls either AEDisposeDesc or your application's token disposal function.

You need to provide a token disposal function only if a call to AEDisposeDesc is not sufficient by itself to dispose of a token or if you provide *marking callback functions* , which are three object callback functions that allow your application to use its own marking scheme rather than tokens when identifying large groups of OSA event objects. Your application is not required to provide marking callback functions.

To handle object specifier records that specify a test, your application must provide two object callback functions: (a) an *object-counting function* , which counts the number of elements of a given object class in a given container so that the OSA Event Manager can determine how many elements it must test to find the element or elements that meet a specified condition, and (b) an *object-comparison function* , which compares one element to another element or to a descriptor record and returns TRUE or FALSE.

Your application may also provide an *error callback function* that can identify which descriptor record caused the resolution of an object specifier record to fail. Your application is not required to provide an error callback function.

If your application resolves object specifier records without the help of the OSA Event Manager, it must extract the equivalent descriptor records and coerce them as necessary to get access to their data. The OSA Event Manager includes coercion handlers for these coercions; for information about this default coercion handling, see the table in section Writing and Installing Coercion Handlers.

For more information about object accessor functions, see Writing Object Accessor Functions. For more information about object callback functions, see Writing Object Callback Functions.

The next section describes how the data in an object specifier record is interpreted by the OSA Event Manager.

Descriptor Records Used in Object Specifier Records

An object specifier record is a coerced AE record of descriptor type typeObjectSpecifier. The data to which its data handle refers consists of four keyword-specified descriptor records:

Keyword	Value	Description of data
keyAEContainer	form	An object specifier record (or in some cases a descriptor record with a handle whose value is NULL) that identifies the container for the requested objects
keyAEDesiredClass	want	A four-character code for the object class
keyAEKeyData	seld	Data or nested descriptor records that specify a property, name, position, range, or test, depending on the key form
keyAEKeyForm	form	A four-character code for the key form

This section describes the descriptor types and data associated with each of these keywords. You need this information if your application resolves or creates object specifier records.

For a summary of the descriptor types and key forms discussed in this section, see the table in section Data Structures Used in Object Specifier Records. For an overview of object specifier records, see Working with Object Specifier Records.

Object Class

The object class of the requested objects is identified by an object class ID. The corresponding keyword-specified descriptor record takes this form:

Keyword	Descriptor type	Data
keyAEDesiredClass	typeType	Object class ID

The *OSA Event Registry: Standard Suites* defines constants for the standard object class IDs.

Container

The container for the requested objects is usually the object in which they are located. It can be identified in one of four ways:

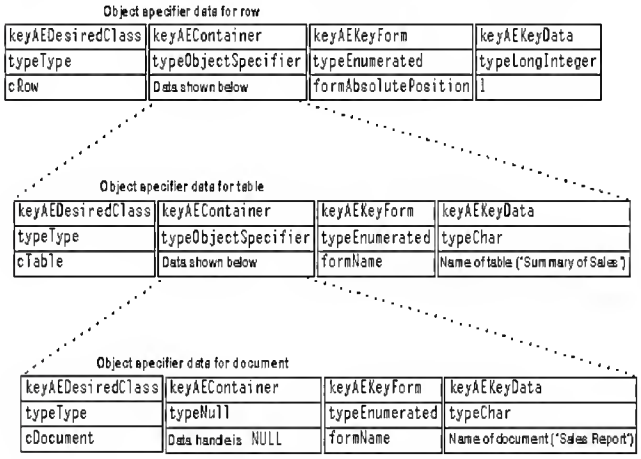
Keyword	Descriptor type	Data
keyAEContainer	typeObjectSpecifier	Object specifier record
	typeNull	Value of data handle is

	NULL. Specifies the default container at the top of the container hierarchy.
typeObjectBeingExamined	Value of data handle is NULL. Specifies the container for elements that are tested one at a time; used only within key data for key form formTest.
typeCurrentContainer	Value of data handle is NULL. Specifies a container for an element that demarcates one boundary in a range. Used only within key data for key form formRange.

The data that describes a container usually consists of another object specifier record. The ability to nest one object specifier record within another in this way makes it possible to identify a chain of containers that fully describes the location of one or more OSA event objects.

For example, the following figure shows nested object specifier records that specify the first row of a table named "Summary of Sales" in a document named "Sales Report." The container specified by the object specifier record at the bottom of the figure describes the outermost container in the container hierarchy-the container for the document "Sales Report."

Because a container must be specified for each OSA event object in a container hierarchy, a *null descriptor record* (that is, a descriptor record whose descriptor type is typeNull and whose data handle has the value NULL) is used to specify the application's default container-the outermost container for any container hierarchy in that application.



Note: The format used in the previous figure and similar figures throughout this chapter does not show the structure of the nested object specifier records as they exist within an OSA event. Instead, these figures show what you would obtain after calling AEGetKeyDesc repeatedly to extract the object specifier records from an OSA event record.

When you call AEGetKeyDesc to extract a null descriptor record, the function returns a descriptor record of type AEDesc with a descriptor type of typeNull and a data handle whose value is 0.

The object specifier data at the bottom of the previous figure uses a null descriptor record to specify the document's container-that is, the default container for the application. The object specifier record for the document identifies the document named "Sales Report"; the object specifier record for the table identifies the table named "Summary of Sales" in the document "Sales Report"; and the object specifier record for the row identifies the first row of the table named "Summary of Sales" in the document "Sales Report."

An object specifier record in an OSA event parameter almost always includes nested object specifier records that specify the container hierarchy for the requested OSA event object. For the nested object specifier records shown in the previous figure, the relationship between each OSA event object and its container is always simple containment: it is located inside its container.

In other cases, the specified container may not actually contain the requested OSA event object. Instead, the relationship between a *container* and a specified object can be defined differently, depending on the key form. For example, the key form formRelativePosition indicates that the requested object is before or after its container.

Object specifier records that specify the key form formTest or formRange require key data that consists of several nested descriptor records, including additional object specifier records that identify either a group of elements to be tested or the boundary elements that demarcate a range. These object specifier records use two special descriptor types to specify containers: typeObjectBeingExamined, which specifies a

container that changes as a group of elements are tested one at a time, and typeCurrentContainer, which specifies the container for a boundary element in a range. Both of these descriptor types require a data handle whose value is NULL, since they act much like variables whose value is supplied by the OSA Event Manager according to other information provided in the container hierarchy.

Key Form

The key form indicates how the key data should be interpreted. It can be specified by one of eight constants:

Keyword	Descriptor type	Data
keyAEKeyForm	typeEnumerated	formPropertyID formName formUniqueID formAbsolutePosition formRelativePosition formTest formWhose formRange

The next section describes the key data that corresponds to each key form.

Key Data

The nature of the information provided by the key data depends both on the specified key form and on the descriptor type of the descriptor record for the key data. The following table summarizes these relationships for the standard key forms.

Key form	Descriptor type	Data
formPropertyID	typeType	Property ID for an element's property
formName	typeChar or other text type	Element's name
formUniqueID	Any appropriate type	A value that uniquely identifies an object within its container or across an application
formAbsolutePosition	typeLongInteger	Offset from beginning (positive) or end (negative) of container
	typeAbsoluteOrdinal	kAEFirst kAEMiddle kAELast kAEAny kAEAll
formRelativePosition	typeEnumerated	kAENext kAEPrevious
formTest	typeCompDescriptor typeLogicalDescriptor	(see tables in section Key Data for a Test)
formRange	typeRangeDescriptor	(see table in section Key Data for a Range)
formWhose	typeWhoseDescriptor	(see table in section Handling Whose Tests)

Most applications that resolve object specifier records need to support only the key forms formPropertyID, formName, formUniqueID, formAbsolutePosition, formRelativePosition, and formRange explicitly. You do not need to support these key forms for all object classes; for example, words usually do not have names, so most applications should return errAEEventNotHandled if they receive a request for a word by name.

If your application provides an object-counting function and an object-comparison function in addition to the appropriate object accessor functions, the OSA Event Manager can handle formTest automatically.

The OSA Event Manager uses the key form formWhose internally to optimize resolution of object specifier records that specify formTest. Applications that translate tests into their own query languages need to support formWhose explicitly. The section Handling Whose Tests describes formWhose in detail.

You can define custom key forms and the format for corresponding data for use by your own application if necessary. If you think you need to do this, check with the OSA Event Registrar first to find out whether existing key forms or others still under development can be adapted to the needs of your application.

One simple kind of key form involves identifying an object on the basis of a specified property. For example, the corresponding data for key form formUniqueID (defined in the *OSA Event Registry: Standard Suites*) always consists of a unique ID for the requested object. This ID is stored as a property identified by the constant pID. The four-character code that corresponds to both formUniqueID and pID is "**ID** ".

If you discover that you do need to define a custom key form based on a property, use the same four-character code for both the key form and the associated property.

The rest of this section describes how the key data for the other key forms shown in The previous table identifies OSA event objects.

Key Data for a Property ID

The key data for formPropertyID is specified by a descriptor record of descriptor type typeType. The *OSA Event Registry: Standard Suites* defines constants for the standard property IDs.

An object specifier record for a property specifies cProperty as the object class ID, an object specifier record for the object that contains the property as the container, formPropertyID as the key form, and a constant such as pFont as the key data. For example, if you were sending a Set Data event to change the font of a word to Palatino, you could specify the data for the object specifier record in the direct parameter as follows:

Keyword	Descriptor type	Data
keyAEDesiredClass	typeType	cProperty
keyAEContainer	typeObjectSpecifier	Object specifier record for word to which property belongs
keyAEKeyForm	typeEnumerated	formPropertyID
keyAEKeyData	typeType	pFont

In this example, the Set Data OSA event parameter identified by the keyword keyAETheData would specify Palatino as the value to which to set the specified property. The reply OSA event for a subsequent Get Data event that included an object specifier record for the same property would return Palatino in the parameter identified by the keyword keyAEResult.

Key Data for an Object's Name

The key data for formName is specified by a descriptor record whose data consists of text, with a descriptor type such as typeChar.

The figure in section Container includes two object specifier records that specify formName.

Key Data for a Unique ID

The key data for formUniqueID consists of a value that identifies an object. This ID must be unique either within the container, at a minimum, or unique across the application. A unique ID can be specified by a descriptor record of any appropriate type; for example, type typeInteger.

Key Data for Absolute Position

The key data for formAbsolutePosition consists of an integer that specifies either an offset or an ordinal position. For descriptor type typeLongInteger, the data is either a positive integer, indicating the offset of the requested element from the beginning of the container, or a negative integer, indicating its offset from the end of the container. The first object specifier record shown in the figure in section Container specifies formAbsolutePosition with key data that consists of the positive integer 1.

For descriptor type typeAbsoluteOrdinal, the data consists of one of these constants:

Constant	Meaning
kAEFirst	First element in the specified container
kAEMiddle	Element in the middle of the specified container
kAELast	Last element in the specified container
kAEAny	Single element chosen at random from the specified container
kAEAll	All the elements in the specified container

If an object specifier record specifies kAEMiddle and the number of elements in the container is even, the OSA Event Manager rounds down; for example, the second word would be the "middle" word in a range of four words.

Key Data for Relative Position

The key data for formRelativePosition is specified by a descriptor record of type typeEnumerated whose data consists of one of these constants:

Constant	Meaning
kAENext	OSA event object after the specified container
kAEPrevious	OSA event object before the specified container

The co*ntainer* can be a single OSA event object or a group of OSA event objects; the requested elements are located immediately before or immediately after it, not inside it.

If your application can locate objects of the same class by absolute position, it can easily locate the same objects by relative position. For example, all applications that support formAbsolutePosition can easily locate the table immediately after a container specified as another table named "Summary of Sales."

Some applications may also be able to locate an object of one class before or after an object of another class. For example, a word processor might be able to locate the paragraph immediately after a container specified as a table named "Summary of Sales."

Key Data for a Test

The key data for formTest is specified by either a comparison descriptor record or a logical descriptor record. If your application provides an object-counting function and an object-comparison function in addition to the appropriate object accessor functions, the OSA Event Manager can handle formTest for you. Some applications may perform tests more efficiently by translating them into the application's own query language. For information about handling tests yourself, see Handling Whose Tests.

The container for objects that pass a test can be one or more OSA event objects. The objects specified are those in the container that pass the test specified by the key data. For example, an object specifier record can describe "the first row in which the First Name column equals 'John' and the Last Name column equals 'Chapman' in the table 'MyAddresses' of the database 'WaveDB.'" To resolve such an object specifier record, the OSA Event Manager must evaluate a logical expression that applies the logical operator AND to two separate comparisons for each row: a comparison of the First Name column to the word "John" and a comparison of the Last Name column to the word "Chapman."

The OSA Event Manager evaluates comparisons and logical expressions on the basis of the information in comparison descriptor records and logical descriptor records. A c*omparison descriptor record* is a coerced AE record of type typeCompDescriptor that specifies an OSA event object and either another OSA event object or data for the OSA Event Manager to compare to the first object. The OSA Event Manager can

also use the information in a comparison descriptor record to compare elements in a container, one at a time, either to an OSA event object or to data. The data for a comparison descriptor record consists of three keyword-specified descriptor records with the descriptor types and data shown in the following table.

Key form	Descriptor type	Data
keyAECompOperator	typeType	kAEGreaterThan kAEGreaterThanEquals kAEEquals kAELessThan kAELessThanEquals kAEBeginsWith kAEEndsWith kAEContains
keyAEObject1	typeObjectSpecifier	Object specifier data
	typeObjectBeingExamined	Value of data handle is NULL
keyAEObject2	typeObjectSpecifier	Object specifier data for object to be compared
	typeObjectBeingExamined	Value of data handle is NULL
	any other type (AEDesc)	Data to be compared

The keyword keyAEObject1 identifies a descriptor record for the element that is currently being compared to the object or data specified by the descriptor record for the keyword keyAEObject2. Either object can be described by a descriptor record of type typeObjectSpecifier or typeObjectBeingExamined. A descriptor record of typeObjectBeingExamined acts as a placeholder for each of the successive elements in a container when the OSA Event Manager tests those elements one at a time. The keyword keyAEObject2 can also be used with a descriptor record of any other descriptor type whose data is to be compared to each element in a container.

You do not have to support all the available comparison operators for all OSA event objects; for example, the "begins with" operator probably does not make sense for objects of type cRectangle. It is up to you to decide which comparison operators are appropriate for your application to support, and how to interpret them. If necessary, you can define your own custom comparison operators. If you think you need to do this, check with the OSA Event Registrar first to find out whether existing definitions of comparison operators or definitions still under development can be adapted to the needs of your application.

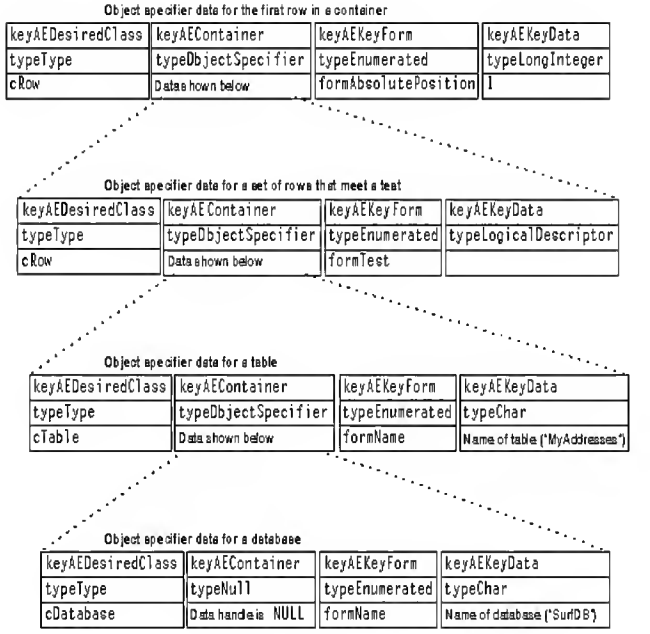
A *logical descriptor record* is a coerced AE record of type typeLogicalDescriptor that specifies a logical expression-that is, an expression that the OSA Event Manager evaluates to either TRUE or FALSE. The logical expression is constructed from a logical operator (one of the Boolean operators AND, OR, or NOT) and a list of logical terms to which the operator is applied. Each logical term in the list can be either another logical descriptor record or a comparison descriptor record. The OSA Event Manager short-circuits its evaluation of a logical expression as soon as one part of the expression fails a test. For example, if while testing a logical expression such as A AND B AND C the OSA Event Manager discovers that A AND B is not true, it will evaluate the expression to FALSE without testing C.

The data for a logical descriptor record consists of two keyword-specified descriptor records with the descriptor types and data shown in the following table.

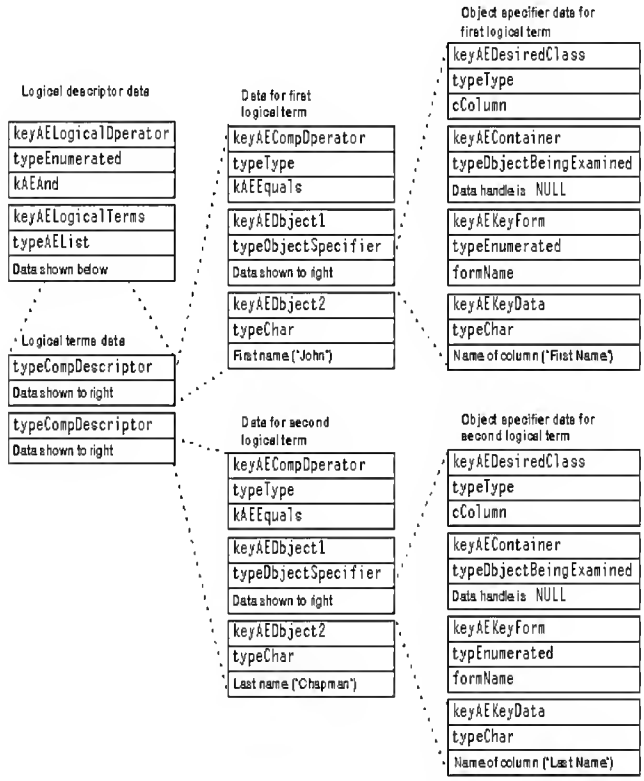
Key form	Descriptor type	Data
keyAELogicalOperator	typeEnumerated	kAEAND kAEOR kAENOT
keyAELogicalTerms	typeAEList	One or more comparison or logical descriptor records

If the logical operator is AND or OR, the list can contain any number of logical terms, and the logical operator is applied to all the terms in the list. For example, the logical descriptor data shown in the second figure below consists of the logical operator AND and a list of logical terms that contains two comparison descriptor records. The entire logical descriptor record corresponds to the logical expression "the First Name column equals 'John' AND the Last Name column equals 'Chapman.'" If the logical operator is NOT, the list must contain a single term.

The following figure shows four object specifier records that specify the container hierarchy for the first row in the table "MyAddresses" of the database "WaveDB" that meets a test. The object specifier record at the top of the following figure specifies the first row contained in the set of rows that form its container. The container for the first row is specified by an object specifier record for a set of rows that meet a test. The two object specifier records at the bottom of the following figure specify the table named "MyAddresses," which contains the rows to be tested, in the database named "WaveDB."



The object specifier record in the previous figure for a set of rows that meet a test specifies formTest. The corresponding key data consists of the logical descriptor record shown in the following figure, which applies the logical operator AND to two logical terms: a comparison descriptor record that specifies all the rows in the container (the table "MyAddresses") in which the column named "First Name" equals "John," and another comparison descriptor record that specifies all the rows in which the column named "Last Name" equals "Chapman." A row in the table "MyAddresses" passes the test only if both comparison descriptor records evaluate as TRUE.



The keyword-specified descriptor records with the keyword keyAEObject1 in the previous figure each consist of an object specifier record that identifies a column by name. The row for each column is specified by a descriptor record of typeObjectBeingExamined, which acts as a placeholder for each row as the OSA Event Manager tests successive rows in the table. The OSA event object specified by each of these object specifier records consists of a column in the row. The OSA Event Manager (with the help of an object-comparison function) compares the contents of the column in successive rows to the string identified by the keyword keyAEObject2 using the comparison operator identified by the keyword keyAECompOperator.

Key Data for a Range

The key data for formRange is specified by a *range descriptor record* , which is a coerced AE record of type typeRangeDescriptor i1.typeRangeDescriptor descriptor type that identifies two OSA event objects marking the beginning and end of a range of elements. The data for a range descriptor record consists of two keyword-specified descriptor records with the descriptor types and data shown in the following table.

Keyword	Descriptor type	Data
keyAERangeStart	typeObjectSpecifier	An object specifier record for the first OSA event object in the desired range
keyAERangeStop	typeObjectSpecifier	An object specifier record for the last OSA event object in the desired range

The elements that identify the beginning and end of the range, which are known as *boundary objects* , do not have to belong to the same object class as the elements in the range itself. If the boundary objects belong to the same object class as the elements in the range, the boundary objects are included in the range. For example, the range of tables specified by boundary elements that are also tables would include the two boundary tables.

The container for boundary objects is usually the same as the container for the entire range, in which case the container for a boundary object can be specified by a placeholder-that is, a descriptor record of type typeCurrentContainer whose data handle has the value NULL.

When AEResolve calls an object accessor function to locate a range of objects, the OSA Event Manager replaces the descriptor record of type typeCurrentContainer with a token for the container of each boundary object. When using AEResolve to resolve the object specifier record, your application does not need to examine the contents of this token, because the OSA Event Manager keeps track of it. If your application attempts to resolve some or all of the object specifier record without calling AEResolve, the application may need to examine the token before it can locate the boundary objects. The token provided by the OSA Event Manager for a boundary object's container is a descriptor record of type typeToken whose data handle refers to a structure of type ccntTokenRecord.

```
typedef struct _ccntTokenRecord {
DescType     tokenClass;       /*Class ID of the container represented by *
                                /* the token*/
AEDesc       token;            /*Token for the current container*/
} ccntTokenRecord;
```

This data type is of interest only if you attempt to resolve an object specifier record for a range without calling AEResolve. Otherwise, the OSA Event Manager keeps track of the container.

Installing Entries in the Object Accessor Dispatch Tables

If the direct parameter for an OSA event consists of an object specifier record, your handler for the event should call the AEResolve function to resolve the object specifier record: that is, to find the OSA event objects or properties it describes. The AEResolve function resolves the object specifier record with the help of object accessor functions provided by your application. Your application installs entries for its object accessor functions in an object accessor dispatch table, which is used by the OSA Event Manager to map requests for OSA event objects or their properties to the appropriate object accessor functions.

After being called by AEResolve, an object accessor function should return a token that identifies (in whatever manner is appropriate for your application) the specified OSA event object or property. An object accessor function also returns a result code that indicates whether it found the OSA event object or property. The token, which is a descriptor record of data type AEDesc, can be of any descriptor type, including descriptor types you define yourself. For an overview of the way AEResolve works with your application's object accessor functions to locate OSA event objects, see Resolving Object Specifier Records.

Each object accessor function provided by your application should either find elements of a specified object class contained in an OSA event object identified by a token of a specified descriptor type, or find properties of an OSA event object identified by a token of a specified descriptor type. To determine which object accessor function to dispatch, the OSA Event Manager uses the object class ID specified in an object specifier record and the descriptor type of the token that identifies the requested object's container. For object accessor functions that find properties, you should specify the object class ID as the constant cProperty.

To install entries in your application's object accessor dispatch table, use the AEInstallObjectAccessor function. For each object class and property your application supports, you should install entries that specify:

- The object class of the requested OSA event object or property
- The descriptor type of the token used to identify the container for the requested OSA event object or property
- The address of the object accessor function that finds objects or properties of the specified object class in containers described by tokens of the specified descriptor type
- A reference constant

You provide this information in the *desiredClass* , *containerType* , *theAccessor* , and *accessorRefcon* parameters to the AEInstallObjectAccessor function.

The *system object accessor dispatch table* is a table that contains object accessor functions available to all processes running on the same computer. The object accessor functions in your application's object accessor dispatch table are available only to your application. If AEResolve cannot find an object accessor function for the OSA event object class in your application's object accessor dispatch table, it looks in the system object accessor dispatch table. If it does not find an object accessor function there either, it returns the result code errAEAccessorNotFound.

If AEResolve successfully calls the appropriate object accessor function in either the application object accessor dispatch table or the system object accessor dispatch table, the object accessor function returns a token and result code. The AEResolve function uses the token and result code to continue resolving the object specifier record. If, however, the token identifies the final OSA event object or property in the container hierarchy, AEResolve returns the token for the final resolution in the *theToken* parameter.

If the AEResolve function calls an object accessor function in the system object accessor dispatch table, your OSA event handler may not recognize the descriptor type of the token returned by the function. If this happens, your handler should attempt to coerce the token to an appropriate descriptor type. If coercion fails, return the result code errAEUnknownObjectType. When your handler returns this result code, the OSA Event Manager attempts to locate a system OSA event handler that can recognize the token.

It is up to you to decide how many object accessor functions you need to write and install for your application. You can install one object accessor function that locates OSA event objects of several different object classes, or you can write separate object accessor functions for certain object classes. Similarly, you may want to use only one descriptor type for all the tokens returned by your object accessor functions, or you may want to use several descriptor types. The sections that follow provide examples of alternative approaches.

For more information about object accessor functions, see Writing Object Accessor Functions.

Installing Object Accessor Functions That Find OSA Event Objects

The following code fragment demonstrates how to add entries to your application's object accessor dispatch table for the object class cText and three of its element classes: the object classes cWord, cItem, and cChar. In this example, the container for each of these object classes is identified by a token that consists of a descriptor record of descriptor type typeMyText.

```
myErr = AEInstallObjectAccessor(cText, typeMyText,
                                (OSLAccessorUPP)&MyFindTextObjectAccessor,
                                0, FALSE);
if (myErr != noErr) DoError(myErr);

myErr = AEInstallObjectAccessor(cWord, typeMyText,
                                (OSLAccessorUPP)&MyFindWordObjectAccessor,
                                0, FALSE);
if (myErr != noErr) DoError(myErr);

myErr = AEInstallObjectAccessor(cItem, typeMyText,
                                (OSLAccessorUPP)&MyFindItemObjectAccessor,
                                0, FALSE);
if (myErr != noErr) DoError(myErr);

myErr = AEInstallObjectAccessor(cChar, typeMyText,
                                (OSLAccessorUPP)&MyFindCharObjectAccessor,
                                0, FALSE);
if (myErr != noErr) DoError(myErr);
```

The first call to AEInstallObjectAccessor in the previous code fragment adds an entry to the application's object accessor dispatch table. This

entry indicates that the AEResolve function should call the MyFindTextObjectAccessor function when resolving any OSA event object with the cText object class and a container identified by a token of descriptor type typeMyText. The other calls to AEInstallObjectAccessor in the previous code fragment add entries for OSA event objects of object classes cWord, cItem, and cChar in a container identified by a token of descriptor type typeMyText. For example, because all the entries created by the code in the previous code fragment specify the descriptor type typeMyText for the token that identifies the container, the AEResolve function calls the MyFindWordObjectAccessor function to locate a requested word regardless of whether the container for the word is a run of text, another word, a paragraph, or an item.

The *accessorRefcon* parameter for the AEInstallObjectAccessor function specifies a reference constant passed to your handler by the OSA Event Manager each time AEResolve calls your object accessor function. Your application can use this reference constant for any purpose. If your application does not use the reference constant, you can use 0 as the value, as shown in the previous code fragment.

If you add an object accessor function to the system object accessor dispatch table, the function that you specify must reside in a DLL. If there was already an entry in the system object accessor dispatch table for the same object class and container descriptor type, that entry is replaced unless you chain it to your system handler. You can do this the same way you chain a previously installed system OSA event handler to your own system handler. See the description of AEInstallEventHandler for details.

The code shown in the previous code fragment installs a separate object accessor function for each object class, even though the code specifies the same descriptor type for tokens that identify the containers for OSA event objects of each class. Most word-processing applications can specify the same object accessor function as well as the same token descriptor type for OSA event objects of these four classes, in which case the code shown in the previous code fragment can be altered as shown in the following code fragment.

```
myErr = AEInstallObjectAccessor(cText, typeMyText,
                                (OSLAccessorUPP)MyFindTextObjectAccessor,
                                0, FALSE);
if (myErr != noErr) DoError(myErr);

myErr = AEInstallObjectAccessor(cWord, typeMyText,
                                (OSLAccessorUPP)MyFindTextObjectAccessor,
                                0, FALSE);
if (myErr != noErr) DoError(myErr);

myErr = AEInstallObjectAccessor(cItem, typeMyText,
                                (OSLAccessorUPP)MyFindTextObjectAccessor,
                                0, FALSE);
if (myErr != noErr) DoError(myErr);

myErr = AEInstallObjectAccessor(cChar, typeMyText,
                                (OSLAccessorUPP)MyFindTextObjectAccessor,
                                0, FALSE);
if (myErr != noErr) DoError(myErr);
```

In some situations, you may want to write different object accessor functions to locate OSA event objects of the same object class in containers identified by tokens of different descriptor types. For example, the code in the following code fragment installs two different object accessor functions: one that finds a word in a container identified by a token of type typeMyTextToken, and one that finds a word in a container identified by a token of typeMyGraphicTextToken.

```
myErr = AEInstallObjectAccessor(cWord, typeMyTextToken,
                                (OSLAccessorUPP)&MyFindTextObjectAccessor,
                                0, FALSE);
if (myErr != noErr) DoError(myErr);

myErr = AEInstallObjectAccessor(cWord, typeMyGraphicTextToken,
                                (OSLAccessorUPP)&MyFindGrphcTextObjectAcces
                                0, FALSE);
if (myErr != noErr) DoError(myErr);
```

Every application must provide one or more object accessor functions that can find OSA event objects in the default container, which is always identified by a token of descriptor type typeNull. The following code fragment demonstrates how to add entries to your application's object accessor dispatch table for the object classes cWindow and cDocument. The container for each of these classes is identified by a token of descriptor type typeNull, which specifies an application's default container.

```
myErr = AEInstallObjectAccessor(cWindow, typeNull,
                                (OSLAccessorUPP)&MyFindWindowObjectAccessor
                                0, FALSE);
```

```
if (myErr != noErr) DoError(myErr);

myErr = AEInstallObjectAccessor(cDocument, typeNull,
                                (OSLAccessorUPP)&MyFindDocumentObjectAccess
                                0, FALSE);
if (myErr != noErr) DoError(myErr);
```

For any entry in your object accessor dispatch table, you can specify a wildcard value for the object class, for the descriptor type of the token used to identify the container, or for both. You specify a wildcard by supplying the typeWildCard constant when installing an entry into the object accessor dispatch table. A wildcard value matches all possible values.

If an object accessor dispatch table contains one entry for a specific object class and a specific token descriptor type, and another entry that is identical except that it specifies a wildcard value for either the object class or the token descriptor type, the OSA Event Manager dispatches the more specific entry. For example, if an object accessor dispatch table includes one entry that specifies the object class as cWord and the token descriptor type as typeMyTextToken, and another entry that specifies the object class as cWord and the token descriptor type as typeWildCard, the OSA Event Manager dispatches the object accessor function associated with the entry that specifies typeMyTextToken.

If you specify typeWildCard as the *desiredClass* parameter and typeMyToken as the *containerType* second parameter for the AEInstallObjectAccessor function and no other entry in the dispatch table matches more exactly, the OSA Event Manager calls the object accessor function that you specify in the *theAccessor* parameter when resolving OSA event objects of any object class in containers identified by tokens of the typeMyToken descriptor type.

If you specify cText as the *desiredClass* parameter and typeWildCard as the *containerType* parameter for the AEInstallObjectAccessor function and no other entry in the dispatch table matches more exactly, the OSA Event Manager calls the object accessor function that you specify in the *theAccessor* parameter when resolving OSA event objects of the object class cText in containers identified by tokens of any descriptor type.

If you specify typeWildCard for both the *desiredClass* and *containerType* parameters of the AEInstallObjectAccessor function and no other entry in the dispatch table matches more exactly, the OSA Event Manager calls the object accessor function that you specify in the *theAccessor* parameter when resolving OSA event objects of any object class in containers identified by tokens of any descriptor type.

Once the OSA Event Manager finds a matching entry, whether exact or involving type typeWildCard, that is the only object accessor function it calls for that object class and token descriptor type. If that function fails, the OSA Event Manager will not look for another matching entry in the same table.

Installing Object Accessor Functions That Find Properties

The OSA event object to which a property belongs is that property's container. You should add entries to your application's object accessor dispatch table that specify object accessor functions for finding properties in containers identified by tokens of various descriptor types. Object specifier records do not specify a property's specific object class; instead, they specify the constant cProperty as the class ID for any property. Similarly, you should specify the constant cProperty as the object class for an object accessor function that can find any property of a container identified by a token of a given descriptor type. If you need to install different object accessor routines for finding properties of OSA event objects that belong to different object classes, you must use different descriptor types for the tokens that represent those OSA event objects.

For example, to specify an object accessor function that locates properties of OSA event objects identified by tokens of descriptor type typeMyToken, you can add a single entry to the object accessor dispatch table:

```
myErr = AEInstallObjectAccessor(cProperty, typeMyToken,
                                (OSLAccessorUPP)MyFindPropertyObjectAccessc
                                0, FALSE);
if (myErr)
  DoError(myErr);
```

The code in this example adds an object accessor function to the application's object accessor dispatch table that can find any property of any container identified by a token of descriptor type typeMyToken. If the *typeMyToken* parameter were specified as typeWildCard, the MyFindPropertyObjectAccessor function would have to be capable of finding any property of any OSA event object in your application except for those found by handlers with more specific entries in the object accessor dispatch table.

Writing Object Accessor Functions

If the direct parameter for an OSA event consists of an object specifier record, your handler for the event should call the AEResolve function to resolve the object specifier record: that is, to find the OSA event objects or properties it describe. The AEResolve function resolves object specifier records with the help of object accessor functions provided by your application. For an overview of the way AEResolve works with your application's object accessor functions to locate OSA event objects, see Resolving Object Specifier Records.

This section describes how to write object accessor functions. You need to read this section if your application supports the Core suite or any of the functional-area suites in the *OSA Event Registry: Standard Suite*s .

Your application should provide object accessor functions that can find OSA event objects and their properties for all object classes supported by your application, including their corresponding properties and element classes. Because the OSA Event Manager dispatches object accessor functions according to the class ID of the requested OSA event object and the descriptor type of the token that identifies its container, you have a great deal of flexibility in deciding what object accessor functions you need to write for your application. The installation and dispatching of object accessor functions are described in Installing Entries in the Object Accessor Dispatch Tables.

For example, if your application is a word processor, one object accessor function will probably work equally well for OSA event objects of object classes cParagraph, cItem, and cWord located in containers identified by tokens of descriptor type myTextToken. If you use a single descriptor type for tokens that identify any containers in which objects of these three object classes can be found, you can dispatch requests for all such elements to the same object accessor function. However, the same word processor might use one descriptor type for tokens identifying containers of class cCell and another descriptor type for tokens identifying containers of class cColumn-in which case it would need an object accessor function for each descriptor type.

For each object class that your application supports, your application should also provide one object accessor function that can find all the properties of that object class, or one object accessor function that can find all the properties of several object classes.

Here is the declaration for a sample object accessor function:

```
OSErr MyObjectAccessor (DescType desiredClass, const AEDesc *containerToken
                        DescType containerClass, DescType keyForm,
                        const AEDesc *keyData, AEDesc *theToken,
                        long *theRefCon)
```

The AEResolve function passes the following information to your object accessor function: the object class ID of the requested OSA event objects, the object class of their container, a token that identifies the specific container in which to look for them, the key form and key data that specify how to locate them, and the reference constant associated with the object accessor function. Your object accessor function uses this information to locate the requested objects.

Most applications that resolve object specifier records need to support only the key forms formPropertyID, formName, formUniqueID, formAbsolutePosition, formRelativePosition, and formRange explicitly. You do not need to support these key forms for all object classes; for example, words usually do not have names, so most applications should return errAEEventNotHandled if they receive a request for a word by name.

If your application provides an object-counting function and an object-comparison function in addition to the appropriate object accessor functions, the OSA Event Manager can handle formTest automatically.

The OSA Event Manager uses the key form formWhose internally to optimize resolution of object specifier records that specify formTest. Only applications that translate tests into their own query languages need to support formWhose explicitly. The section Handling Whose Tests describes formWhose in detail.

If your object accessor function successfully locates the requested OSA event objects, your application should return the noErr result code and a token that identifies them. The token can be of any descriptor type, as long as it is a descriptor record. For example, to identify a file, your application might use a descriptor record of descriptor type typeOS2FileName. To identify an open document, your application might define its own descriptor type, such as typeMyDocToken, for a descriptor record whose data handle refers to a pointer to a document record. For more information about tokens, see Defining Tokens.

Important: Object accessor functions must not have side effects that change the number or order of elements in a container while an object specifier record is being resolved. If the number of elements in a container is changed during the resolution of an object specifier record, the OSA Event Manager may not be able to locate all the elements.

Writing Object Accessor Functions That Find OSA Event Objects

The first three listings in this section demonstrate how to write three object accessor functions that might be called in the following situation: An application receives a Get Data event with a direct parameter that consists of an object specifier record for the first word in the third paragraph of a document. The application's handler for the Get Data event calls the AEResolve function to resolve the object specifier record. The AEResolve function first calls the application's object accessor function for objects of class cDocument in containers identified by a token of descriptor type typeNull.

The AEResolve function passes these values to the MyFindDocumentObjectAccessor function shown in the following code fragment: in the *desiredClass* parameter, the constant cDocument; in the *containerToken* parameter, a descriptor record of descriptor type typeNull with a data handle whose value is NULL; in the *containerClass* parameter, the constant typeNull; in the *keyForm* parameter, the constant formName; in the *keyData* parameter, a descriptor record of descriptor type typeText whose data consists of the string "MyDoc"; and the reference constant specified in the application's object accessor dispatch table.

```
OSErr MyFindDocumentObjectAccessor (DescType desiredClass,
                                    const AEDesc *containerToken,
                                    DescType containerClass, DescType keyFc
                                    const AEDesc *keyData, AEDesc *theToken
                                    long *theRefCon)
{

    char     docName[NAMESIZE];
    Size     actSize;
    BOOL     foundDoc;
    Document *foundDocRecPtr;

    switch (keyForm) {
      case formName:
          /* get the name of the document from the key data */
          MyGetStringFromDesc(keyData, docName, &actSize);
          /* look for a document with the given name by */
          /* searching all document records */
          MySearchDocRecs(docName, foundDocRecPtr, &foundDoc);
          if (!foundDoc) {
            return (kObjectNotFound);
          }
          else {
            /* create token that identifies the document */
            myErr = AECreateDesc(typeMyDocToken, &foundDocRecPtr,
                                 sizeof(foundDocRecPtr), &token);
            return (myErr);
          }
          break;

      default:
          /* handle the other key forms you support */
          return (kKeyFormNotSupported);
          break;
  }
}
```

The MyFindDocumentObjectAccessor function uses the information in the keyForm and *keyData* parameters to find the specified document. If it finds the OSA event object, MyFindDocumentObjectAccessor returns a token of descriptor type typeMyDocToken to AEResolve. The data handle for this token refers to a pointer to a document record (see the figure in section Defining Tokens). The MyFindDocumentObjectAccessor function returns this token and the noErr result code to the AEResolve function.

In the Get Data example, the token returned to AEResolve by the MyFindDocumentObjectAccessor function identifies the document "MyDoc." The AEResolve function then calls the application's object accessor function for objects of class cParagraph in containers identified by a token of descriptor type typeMyDocToken.

In this case, AEResolve passes these values to the MyFindParaObjectAccessor function shown in the following code fragment: in the *desiredClass* parameter, the constant cParagraph; in the *containerToken* parameter, the token returned by the MyFindDocumentObjectAccessor function; in the *containerClass* parameter, the constant cDocument; in the *keyForm* parameter, the constant formAbsolutePosition; in the *keyData* parameter, a descriptor record with the typeLongInteger descriptor type and data that consists of the value 3 (indicating the third paragraph); and the reference constant specified in the application's object accessor dispatch table.

```
OSErr MyFindParaObjectAccessor (DescType desiredClass,
                                const AEDesc *containerToken,
```

```
                                    DescType containerClass, DescType keyForm,
                                    const AEDesc *keyData, AEDesc *theToken,
                                    long *theRefCon)
{
    long                index;
    /*MyFoundTextRecord is an application-defined data type */
    /* consisting of three fields: start, ending, and docPtr*/
    MyFoundTextRecord foundParaRec;
    long                foundParaStart;
    long                foundParaEnd;
    Document            *foundDocRecPtr;
    BOOL                success;

    switch (keyForm) {
        case formAbsolutePosition:
          /*get the index of the paragraph from the key data*/
          MyGetIndexFromDesc(keyData, index);
          /*get the desired paragraph by index*/
          success = MyGetPara(index, containerToken, foundParaStart,
                              foundParaEnd, foundDocRecPtr);
          if (!success)
            return (kObjectNotFound);
          else { /* create token that identifies the paragraph */
            foundParaRec.start = foundParaStart;
            foundParaRec.ending = foundParaEnd;
            foundParaRec.docPtr = *foundDocRecPtr;
            myErr = AECreateDesc(typeMyTextToken, &foundParaRec,
                                 sizeof(foundParaRec), &token);
            return (myErr);
          }
          break;

      default:
          /* handle the other key forms you support */
          return (kKeyFormNotSupported);
          break;
    }
}
```

The MyFindParaObjectAccessor function uses another application-defined function, MyGetPara, to search the data structures associated with the document and find the desired paragraph. If it finds the paragraph, MyGetPara returns a value that identifies the beginning of the paragraph, a value that identifies the end of the paragraph, and a pointer to the document (which MyGetPara gets from the *containerToken* parameter). The MyFindParaObjectAccessor function returns an application-defined token that contains this information. This token is of descriptor type typeMyTextToken; it describes a range of characters that can be used to identify any range of text, including a paragraph or a word. The MyFindParaObjectAccessor function returns this token and the noErr result code to the AEResolve function.

In the Get Data example, the token returned to AEResolve by the MyFindParaObjectAccessor function identifies the third paragraph in the document "MyDoc." The AEResolve function then calls the application's object accessor function for objects of class cWord in containers identified by a token of descriptor type typeMyTextToken.

In this case, the AEResolve function passes these values to the MyFindWordObjectAccessor function shown in the following code fragment: in the *desiredClass* parameter, the constant cWord; in the *containerToken* parameter, the token returned by the MyFindParaObjectAccessor function (a token of descriptor type typeMyTextToken that identifies a paragraph); in the *containerClass* parameter, the constant cParagraph; in the *keyForm* parameter, the constant formAbsolutePosition; in the *keyData* parameter, a descriptor record with the typeLongInteger descriptor type and data that consists of the value 1 (indicating the first word); and the reference constant specified in the application's object accessor dispatch table.

The MyFindWordObjectAccessor function uses another application-defined function, MyGetWord, to search the paragraph to find the desired word. If it finds the word, MyGetWord returns a value that identifies the beginning of the word, a value that identifies the end of the word, and a pointer to the document (which MyGetWord gets from the *containerToken* parameter). The MyFindWordObjectAccessor function returns a token that contains this information. This token is also of descriptor type typeMyTextToken; in this case, the token identifies a specific word.

The MyFindWordObjectAccessor function returns this token and the noErr result code to the AEResolve function, which in turn returns the token to the Get Data event handler that originally called AEResolve.

```
OSErr MyFindWordObjectAccessor (DescType desiredClass,
                                const AEDesc *containerToken,
                                DescType containerClass, DescType keyForm,
                                const AEDesc *keyData, AEDesc *theToken,
                                long *theRefCon)
{
    long                index;
    MyFoundTextRecord foundWordRec;
    long                foundWordStart;
    long                foundWordEnd;
    Document            *foundDocRecPtr;
    BOOL                success;

    switch (keyForm) {
      case formAbsolutePosition:
          /*get the index of the word from the key data*/
          MyGetIndexFromDesc(keyData, index);
          /*get the desired word by index*/
          success = MyGetWord(index, containerToken, foundWordStart,
                              foundWordEnd, foundDocRecPtr);
          if (!success)
            return (kObjectNotFound);
          else {
            /*create token that identifies the paragraph*/
            foundWordRec.start = foundWordStart;
            foundWordRec.ending = foundWordEnd;
            foundWordRec.docPtr = *foundDocRecPtr;
            myErr = AECreateDesc(typeMyTextToken, &foundWordRec,
                                 sizeof(foundWordRec), &token);
          }
          break;
      default:
        /*handle the other key forms you support*/
        return (kKeyFormNotSupported);
  }
}
```

The first code fragment in this section shows an object accessor function that locates a document in the default container. Every application must provide one or more object accessor functions that can find OSA event objects in the default container, which is always identified by a descriptor record of descriptor type typeNull. The following code fragment provides another example of an object accessor function that locates an OSA event object in the default container. If the MyFindWindowObjectAccessor function shown in the following code fragment were installed in an application's object accessor dispatch table, the AEResolve function would call it as necessary to locate an object of class cWindow in a container identified by a token of descriptor type typeNull.

```
OSErr APIENTRY MyFindWindowObjectAccessor(DescType desiredClass,
                                          const AEDesc *containerToken,
                                          DescType containerClass,
                                          DescType keyForm,
                                          const AEDesc *keyData,
                                          AEDesc *theToken,
                                          long *theRefCon)
{
  OSErr myErr = noErr;
  BOOL found = FALSE;
```

```
  CHAR windowName[CCHMAXPATH], windowTitle[CCHMAXPATH];
LONG index;
HWND hwnd;
Size actualSize;

switch(keyForm)
{
    case formName:
      /* Find the window given a name in key data */
      /* Get the window name out of key data */
      MyGetStringFromDesc(keyData, windowName, &actualSize);
      /* look for a window with the given name */
      hwnd = MyGetFrontWindow();
      while( (hwnd != NULL) && (found == FALSE) )
      {
          MyGetWindowTitle(hwnd, windowTitle);
          if(strcmp(windowName, windowTitle) == 0)
          {
            found = TRUE;
          }
          else
          {
            hwnd = MyGetNextWindow(hwnd);
          }
      }
      break;
    case formAbsolutePosition:
      /* Find the window given an index in key data */
      /* Get the index out of key data */
      MyGetIndexFromDesc(keyData, index);
      /* look for a window with the given index */
      hwnd = MyGetFrontWindow();
      while( (hwnd != NULL) && (found == FALSE) )
      {
          MyGetWindowIndex(hwnd, &windowIndex);
          if(index == windowIndex)
          {
            found = TRUE;
          }
          else
          {
            hwnd = MyGetNextWindow(hwnd);
          }
      }
      break;
    /* Handle the other key forms you support */
    default:
      myErr = kKeyFormNotSupported;
      break;
}

if(myErr == noErr)
{
    if(found)
    {
```

```
            myErr = AECreateDesc(typeMyWindow, &hwnd, sizeof(hwnd), theToken);
        }
        else
        {
            myErr = kObjectNotFound;
        }
    }
    return myErr;
}
```

The *keyForm* parameter of the MyFindWindowObjectAccessor function describes how the function should interpret the *keyData* parameter. If the key form is formName, then the key data contains the name of the window to locate. If the key form is formAbsolutePosition, the key data contains the position of the window to locate in the window list; for example, a value of 1 identifies the front most window.

The MyFindWindowObjectAccessor function supports only the formName and formAbsolutePosition key forms. Your object accessor functions should support all key forms that make sense for the kinds of objects the functions can locate.

For the formName keyword, the MyFindWindowObjectAccessor function starts with the frontmost window and compares the window's title to the name specified by the *keyData* parameter. It continues this search until it reaches either the end of the window list or finds a match. If the MyFindWindowObjectAccessor function finds a match, it uses the AECreateDesc function to create a descriptor record for the token, specifying the application-defined typeMyWindow descriptor type and the data for this descriptor type as a window pointer.

The MyFindWindowObjectAccessor function then sets its function result appropriately, and the AEResolve function either returns this function result and token, or uses the returned token to request the next OSA event object in the container hierarchy, such as a document in the window.

Writing Object Accessor Functions That Find Properties

The OSA event object to which a property belongs is that property's container. Your application should provide an object accessor function for finding properties in containers identified by tokens of various descriptor types. Your application does not need to be given a property's specific object class in order to find that property; instead, you can specify the object class ID for any property with the constant cProperty. Thus, you can write a single object accessor function that can find any property of an object identified by a token of a given descriptor type.

To install such an object accessor function, you can add a single entry to the object accessor dispatch table that specifies the desired object class as cProperty for a given token descriptor type. For example, the following code fragment shows an object accessor function that identifies any property of a window.

```
OSErr MyFindPropertyOfWindowObjectAccessor (DescType desiredClass,
                                            const AEDesc *containerToken,
                                            DescType containerClass,
                                            DescType keyForm,
                                            const AEDesc *keyData,
                                            AEDesc *theToken, long *theRefC
{

    DescType theProperty;
    OSErr    myErr;

    myErr = noErr;
    MyGetPropFromKeyData(keyData, theProperty);
    switch (keyForm) {
        case fromPropertyID:
          switch (theProperty) {
              case pName:
                /* create token that identifies name property of the window
                MyCreateToken(typeMyWindowProp, containerToken, pName, toke
                break;
              case pBounds:
```

```
                /* create token that identifies bounds property of the */
                /* window */
                MyCreateToken(typeMyWindowProp, containerToken, pBounds, tc
                /* create tokens for other properties as appropriate */
                break;
            default:
                myErr = kErrorPropNotFound;
                break;
        }
        break;
    default:
        myErr = kKeyFormNotSupported;
        break;
  }
  return (myErr);
}
```

The MyFindPropertyOfWindowObjectAccessor function takes a token that identifies a window and creates a token that identifies the requested property of that window. See the second figure in section Defining Tokens for an illustration of the logical organization of a token of descriptor type typeMyWindowProp.

This simplified example merely translates information about the requested property and the window to which it belongs into the form of a token of type typeMyWindowProp. This token can then be used by OSA event handlers to identify the corresponding window and its property, so that a handler can either retrieve the value of the property (for example, a Get Data handler) or change the value of the property (for example, a Set Data handler). Like other tokens, a token that identifies a property should always contain a reference to the corresponding property and the object to which it belongs-not a copy of the data for that object's property.

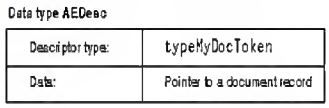
Defining Tokens

It is up to you to decide how many token descriptor types you need to define for your application. In many cases you may be able to define one token that can identify OSA event objects of several different object classes, such as a token of type typeMyTextToken that identifies OSA event objects of object classes cText, cWord, cItem, and cChar. In other cases you may need to define specific token descriptor types for specific object classes.

For example, the MyFindDocumentObjectAccessor routine shown in the first figure in section Writing Object Accessor Functions That Find OSA Event Objects returns a token of descriptor type typeMyDocToken, which identifies a document record.

```
/* application-defined token */
#define   typeMyDocToken   0x72636F64   /* 'docr' */
```

The following figure shows the logical arrangement of a descriptor record of descriptor type typeMyDocToken whose data is specified by a pointer to a document record.



The MyFindPropertyOfWindowObjectAccessor routine shown in the figure in section Writing Object Accessor Functions That Find Properties returns a token of descriptor type typeMyWindowProp for every property that it can locate.

```
/* application-defined token */
#define typeMyWindowProp     0x726C6F63   /* 'colr' */
```

The following figure shows the logical arrangement of a descriptor record of descriptor type typeMyWindowProp that identifies the bounds property of a window. Its data consists of a window pointer and the constant pBounds. The application can use this token either to return or to change the window's bounds setting, depending on the OSA event that specified the property. If the token specified pName instead, the application could use it either to return the window's name as a string or to change the window's name.

Data type AEDesc

Descriptor type:	typeMyWindowProp
Data:	Window pointer
	pBounds

A token's data should always contain a reference to the corresponding OSA event objects-not a copy of the data for those objects. This allows the same token to be used for both reading and writing tokens.

It is often possible to use the same token type for objects of several object classes, or for both an object of a given class and one of its properties. A token's data is private to your application and can be organized in any way that is convenient.

When an object accessor function that supports key form formRange locates a range of OSA event objects, it should normally return a descriptor list (AEDescList) of tokens for the individual objects. A typical exception is an object accessor function that returns a range of objects of class cText, which should return a single token representing the entire range. For example, an object accessor function that finds "all the characters from char 1 to char 1024" should return a token that consists of a list of 1024 objects, each of class cChar, whereas an object specifier function that finds "all the text from char 1 to char 1024" should return a single token for a single item of class cText that is 1024 characters long.

A token is valid only until the OSA Event Manager has located the requested element in the container the token represents and returned another token for the element. The OSA Event Manager disposes of intermediate tokens after it finishes resolving an object specifier record, but one token is always left over-the token that identifies the specified OSA event object or objects. Your application should dispose of this final token by calling the AEDisposeToken function, which in turn calls your application's token disposal function (if one exists), an optional object callback function that disposes of a token.

If your application does not provide a token disposal function, the OSA Event Manager uses the AEDisposeDesc function to dispose of tokens. This function does the job as long as disposing of tokens involves nothing more than simply disposing of a descriptor record. Otherwise, you need to provide a custom token disposal function. For example, suppose the data field of a token descriptor record contains a handle to a block that in turn contains references to memory for the OSA event object referred to by the token. In this case, the application should provide a token disposal function that performs the tasks required to dispose of the token and any associated structures.

Handling Whose Tests

If your application provides an object-counting function and an object-comparison function in addition to the appropriate object accessor functions, the OSA Event Manager can resolve object specifier records that specify formTest without any other assistance from your application. The OSA Event Manager translates object specifier records of key form formTest into object specifier records of key form formWhose. This involves collapsing the key form and key data from two object specifier records in a container hierarchy into one object specifier record with the key form formWhose.

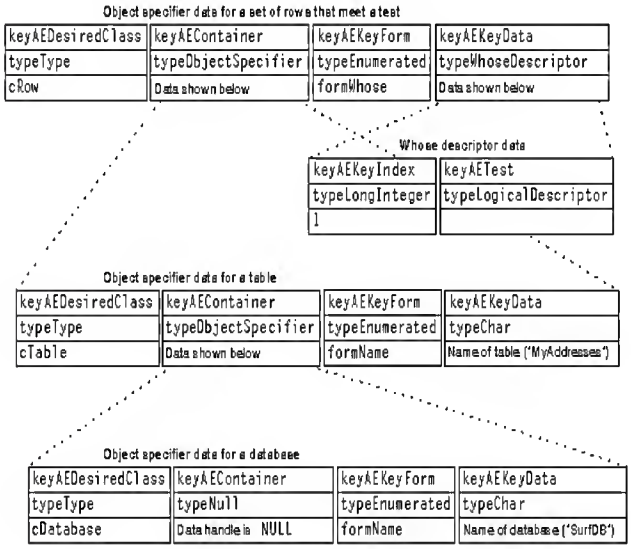
Some applications may find it more efficient to translate whose tests into their own query languages rather than letting the OSA Event Manager handle the tests. This is useful only for applications that can make use of a test combined with either an absolute position or a range to locate objects. If you want the OSA Event Manager to let your application handle whose tests, set the kAEIDoWhose flag in the c*allbackFlag*s parameter of the AEResolve function. If for any reason one of your application's object accessor functions chooses not to handle a particular whose descriptor record, it should return errAEEventNotHandled as the result code, and the OSA Event Manager will try again using the original object specifier records, just as if the kAEIDoWhose flag were not set.

The key data for formWhose is specified by a *whose descriptor record* , which is a coerced AE record of descriptor type typeWhoseDescriptor. The data for a whose descriptor record consists of the two keyword-specified descriptor records shown in the following table.

Keyword	Descriptor type	Data
keyAEIndex	typeLongInteger	Offset of requested element in group of elements that pass a test
	typeAbsoluteOrdinal	kAEFirst kAEMiddle kAELast kAEAny kAEAll
	typeWhoseRange	Whose range descriptor record
keyAETest	typeCompDescriptor	Comparison descriptor record
	typeLogicalDescriptor	Logical descriptor record

A whose descriptor record is never created directly by an application. The OSA Event Manager creates a whose descriptor record whenever an object specifier record of key form formTest is used to describe the container for elements described by an object specifier record of key form formAbsolutePosition or formRange, with some exceptions as noted in this section.

For example, the first figure in section Key Data for a Test shows four object specifier records that show the container hierarchy for the first row that meets a test in the table "MyAddresses" of the database "WaveDB." The top two object specifier records in that figure use the key forms formAbsolutePosition and formTest to describe elements in a container. When it receives these two object specifier records, the OSA Event Manager collapses them into one, as shown in the following figure. It then calls the application's object-counting function to find out how many objects of class cRow the table contains and the object-comparison function to test the rows in the table until it finds the first row that passes the test.



If the elements to be tested are described by an object specifier record of key form formAbsolutePosition or formRange but are not of the same object class as their container, the OSA Event Manager cannot collapse the existing object specifier records into a whose descriptor record. Instead, the OSA Event Manager creates a whose descriptor record as if a third object specifier record of key form formAbsolutePosition and kAEAll were inserted between the object specifier record for the container and that for the tested elements. For example, the OSA Event Manager would interpret a request for "character 1 of word whose first letter = 'a'" as "character 1 of every word whose first letter = 'a'."

When an object specifier record of key form formTest is used to describe the container for elements described by an object specifier record of key form formRange, the OSA Event Manager will, under certain conditions, coerce the corresponding range descriptor record to a whose range descriptor record, which is a coerced AE record of typeWhoseRange. The data for a *whose range descriptor record* consists of two keyword-specified descriptor records with the descriptor types and data shown in the following table.

Keyword	Descriptor type	Data
keyAEWhoseRangeStart	typeLongInteger	Offset of beginning of range
	typeAbsoluteOrdinal	kAEFirst kAEMiddle kAELast kAEAny kAEAll
keyAEWhoseRangeStop	typeLongInteger	Offset of end of range
	typeAbsoluteOrdinal	kAEFirst kAMiddle kAELast kAEAny kAEAll

A whose range descriptor record describes the absolute position of the boundary elements, within the set of all elements that pass a test, that identify the beginning and end of the desired range.

The OSA Event Manager coerces a range descriptor record to a whose range descriptor record if the specified container and its elements are of the same class, if the container for the specified range of elements is a group of OSA event objects that pass a test, and if the boundary objects in the original range descriptor record meet these conditions:

- Both boundary objects are of the same object class as the OSA event objects in the range they specify.

- The object specifier record for each boundary object specifies its container with a descriptor record of descriptor type typeCurrentContainer.
- The object specifier record for each boundary object specifies a key form of formAbsolutePosition.

If these conditions are not met, the OSA Event Manager does not create a whose range descriptor record. Instead, as described earlier in this section, the OSA Event Manager creates a whose descriptor record as if the original request specified every element that passed the test.

If your application sets the kAEIDoWhose flag in the *callbackFlags* parameter of AEResolve, you should provide object accessor functions that can handle formWhose. These functions should coerce the whose descriptor record specified as key data for an object specifier record to an AE record and extract the data from the AE record by calling the AEGetKeyPtr and AEGetKeyDesc functions. If the keyword-specified descriptor record with the keyword keyAEIndex specifies descriptor type typeWhoseRange, your object accessor function must also coerce that descriptor record to an AE record and extract the data. Your object accessor function should then attempt to locate the requested objects and, if successful, return a token that identifies them.

If your application sets the kAEIDoWhose flag and attempts to resolve every whose descriptor record it receives, the OSA Event Manager does not attempt to resolve object specifier records of any key form. The object-counting and object-comparison functions are never called, and your application is solely responsible for determining the formats and types of all tokens.

Writing Object Callback Functions

If an OSA event parameter consists of an object specifier record, your handler for the OSA event typically calls AEResolve to begin the process of locating the requested OSA event object or objects. In turn, AEResolve calls object accessor functions and, if necessary, object callback functions provided by your application.

Every application that supports OSA event objects should provide object accessor functions that can locate OSA event objects belonging to any of the supported object classes. For an overview of the way AEResolve calls object accessor functions to locate OSA event objects described by object specifier records, see Resolving Object Specifier Records.

In addition to object accessor functions, your application can provide up to seven object callback functions:

- An *object-counting function* counts the number of elements of a specified class in a specified container, so that the OSA Event Manager can determine how many elements it must examine to find the element or elements that pass a test. Your application must provide one object-counting function to handle object specifier records that specify tests. (See Writing an Object-Counting Function.)
- An *object-comparison function* compares one element either to another element or to a descriptor record and returns either TRUE or FALSE. Your application must provide one object-comparison function to handle object specifier records that specify tests. (See Writing an Object-Comparison Function.)
- A *token disposal function* disposes of a token after your application calls the AEDisposeToken function. If your application does not provide a token disposal function, the OSA Event Manager uses the AEDisposeDesc function instead. Your application must provide a token disposal function if it requires more than a call to AEDisposeDesc to dispose of one of its tokens. This is true, for example, if your application supports marking by modifying its own data structures.
- An *error callback function* gives the OSA Event Manager an address to which to write the descriptor record it is currently working with if an error occurs while AEResolve is attempting to resolve an object specifier record. Your application is not required to provide an error callback function.
- Three *marking callback functions* are used by the OSA Event Manager to get a mark token from your application, to mark specific OSA event objects, and to pare down a group of marked OSA event objects. Your application must provide all three marking functions if it supports marking. (See Writing Marking Callback Functions.)

To make your object callback functions available to the OSA Event Manager, use the AESetObjectCallbacks function:

```
myErr = AESetObjectCallbacks ((OSLCompareUPP)MyCompareObjects,
                              (OSLCountUPP)MyCountObjects,
                              (OSLDisposeUPP)MyDisposeToken,
                              (OSLGetMarkUPP)MyGetMarkToken,
                              (OSLMarkUPP)MyMark,
                              (OSLAdjustMarksUPP)MyAdjustMarks,
                              (OSLGetErrUPP)MyGetErrDesc);
```

Each parameter to the AESetObjectCallbacks function consists of either a pointer to the corresponding application-defined function or NULL if no function is provided. The AESetObjectCallbacks function sets object callback functions that are available only to your application. To set

system object callback functions, which are available to all applications and processes running on the same computer, use the AEInstallSpecialHandler function.

To handle object specifier records that specify tests, your application must provide an object-counting function and an object-comparison function. The OSA Event Manager calls your application's object-counting function to determine the number of OSA event objects in a specified container that need to be tested. The OSA Event Manager calls your application's object-comparison function when it needs to compare one OSA event object to either another OSA event object or to a value in a descriptor record.

If your application does not provide a token disposal function, the OSA Event Manager uses the AEDisposeDesc function to dispose of tokens. This function does the job as long as disposing of tokens involves nothing more than simply disposing of a descriptor record. Otherwise, you need to provide custom token disposal function. For example, suppose the data field of a token descriptor record contains a handle to a block that in turn contains references to storage for the OSA event object referred to by the token. In this case, the application can provide a token disposal function that performs the tasks required to dispose of the token and any associated structures.

Whenever more than one OSA event object passes a test, AEResolve can either return a list of tokens or make use of a target application's ability to mark its own objects. Sometimes a list of tokens can become unmanageably large. For example, if a Get Data event asks for the names and addresses of all customers with a specified zip code who have purchased a specified product, the object accessor function that locates all the customers with the specified zip code might return a list of many thousands of tokens; the elements identified by those tokens would then have to be tested for the specified product. However, if your application uses some method of marking objects, you can choose simply to mark the requested objects rather than returning a list of tokens. Writing Marking Callback Functions describes how to do this. If your application supports marking by modifying its own data structures, you must provide a token disposal function.

When one of your application's OSA event handlers calls the AEResolve function, the handler should pass a value in the *callbackFlags* parameter that specifies whether your application supports whose descriptor records or provides marking callback functions. You can add the following constants, as appropriate, to provide a value for the *callbackFlags* parameter:

```
#define kAEIDoMinimum    0x0000;    /*Does not handle whose tests or */
                                    /* provide marking callbacks*/
#define kAEIDoWhose      0x0001;    /*Supports key form formWhose*/
#define kAEIDoMarking    0x0004;    /*Provides marking functions*/
```

For example, this code instructs the OSA Event Manager to call any marking functions previously set with the AESetObjectCallbacks function while resolving the object specifier record in the *objectSpecifier* parameter:

```
AEDesc objectSpecifier;
AEDesc resultToken;
OSErr   myErr;

myErr = AEResolve(&objectSpecifier, kAEIDoMarking, &resultToken);
```

If any of the marking callback functions are not installed, AEResolve returns the error errAEHandlerNotFound.

Important: If your application does not specify kAEIDoWhose, the OSA Event Manager attempts to resolve all object specifier records of key form formTest. To do so, the OSA Event Manager uses your application's object-counting and object-comparison functions, and returns a token of type typeAEList.

If your application does specify kAEIDoWhose, the OSA Event Manager does not attempt to resolve object specifier records of any key form. In this case, the object-counting and object-comparison functions are never called; your application determines the formats and types of all tokens; and your application must interpret whose descriptor records created by the OSA Event Manager during the resolution of object specifier records. For more information, see Handling Whose Tests.

Writing an Object-Counting Function

To handle object specifier records that specify tests, your application should provide an object-counting function (unless it specifies kAEIDoWhose as just described). Your object-counting function should be able to count the number of elements of a given object class in a given container. For example, if your application supports OSA event objects that belong to the object class cText in the Text suite, your application should provide an object-counting function that can count OSA event objects of each element class listed in the definition of cText in the *OSA Event Registry: Standard Suites* . In this case, your application should provide an object-counting function that can count the number of words, items, or characters in a text object.

You specify your object-counting function with the AESetObjectCallbacks function. Whenever it is resolving an object specifier record and it

requires a count of the number of elements in a given container, the OSA Event Manager calls your object-counting function.

Here is the declaration for a sample object-counting function:

```
OSErr MyCountObjects (DescType desiredClass, DescType containerClass,
                      const AEDesc *containerToken, long *result);
```

The OSA Event Manager passes the following information to your object-counting function: the object class ID of the OSA event objects to count, the object class of their container, and a token identifying their container. (The container class can be useful if you want to use one token type for several object classes.) Your object-counting function uses this information to count the number of OSA event objects of the specified object class in the specified container. After counting the OSA event objects, your application should return the noErr result code and, in the *result* parameter, the number of OSA event objects counted.

The following code fragment shows an application-defined function, MyCountObjects, that counts the number of objects for any object class supported by the application.

```
OSErr APIENTRY MyCountObjects(DescType desiredClass,
                              DescType containerClass,
                              const AEDesc *containerToken,
                              long *result)
{
  OSErr myErr = noErr;

  *result = 0;

  switch(desiredClass)
  {
      case cWindow:
        if(containerClass == typeNull)
        {
            /* count the number of windows */
            HWND hwnd = MyGetFrontWindow();
            while( hwnd != NULL)
            {
              ++result;
              hwnd = MyGetNextWindow(hwnd);
            }
        }
        break;
      case cWord:
        /* count the number of words in the container */
        myErr = MyCountWords(containerClass, containerToken, result);
        break;
      case cParagraph:
        /* count the number of paragraphs in the container */
        myErr = MyCountParagraphs(containerClass, containerToken, result);
        break;
      default:
        /* this app does not support any other object classes */
        myErr = kObjectClassNotFound;
        break;
  }
  return myErr;
}
```

Writing an Object-Comparison Function

To handle object specifier records that specify tests, your application should provide an object-comparison function (unless it specifies kAEIDoWhose). Your object-comparison function should be able to compare one OSA event object to another OSA event object or to another descriptor record.

You specify your object-comparison function with the AESetObjectCallbacks function. Whenever it is resolving object specifier records and needs to compare the value of an OSA event object with another object or with data, the OSA Event Manager calls your object-comparison function.

Here is the declaration for a sample object-comparison function:

```
OSErr MyCompareObjects (DescType comparisonOperator, const AEDesc *theObjec
                        AEDesc *objectOrDescToCompare, BOOL *result);
```

The OSA Event Manager passes the following information to your object-comparison function: a comparison operator that specifies how the two objects should be compared, a token for the first OSA event object, and either a token that describes the OSA event object to compare or a descriptor record.

It is up to your application to interpret the comparison operators it receives. The meaning of comparison operators differs according to the OSA event objects being compared, and not all comparison operators apply to all object classes. After successfully comparing the OSA event objects, your object-comparison function should return the noErr result code and, in the *result* parameter, a Boolean value specifying TRUE if the result of the comparison is true and FALSE otherwise. If for any reason your comparison function is unable to compare the specified OSA event objects, it should return the result code errAEEventNotHandled; then the OSA Event Manager will try an alternative method of comparing the OSA event objects, such as calling the equivalent system object-comparison function, if one exists.

Your object-comparison function should be able to compare an OSA event object belonging to any object class with another OSA event object. Your function should also be able to compare two OSA event objects with different object classes, if appropriate. For example, an object-comparison function for a word-processing application might be asked to compare the First Name column of a specified row in a table with the first word on a specified page-that is, to compare an OSA event object of object class cColumn with an OSA event object of object class cWord. You must decide what kinds of comparisons make sense for your application.

The *OSA Event Registry: Standard Suite*s defines standard comparison operators. Here is a list of the constants that correspond to these comparison operators:

```
#define kAEBeginsWith              0x74776762   /* "bgwt" */
#define kAEContains                0x746E6F63   /* "cont" */
#define kAEEndsWith                0x73646E65   /* "ends" */
#define kAEEquals                  0x2020203D   /* "=   " */
#define kAEGreaterThan              0x2020203E   /* ">   " */
#define kAEGreaterThanEquals        0x20203D3E   /* "=> " */
#define kAELessThan                0x2020203C   /* "<   " */
#define kAELessThanEquals          0x20203D3C   /* "=< " */
```

The comparison operators always relate the first operand to the second. For example, the constant kAEGreaterThan means that the object-comparison function should determine whether or not the value of the first operand is greater than the value of the second operand. For more information, see AECreateCompDescriptor function.

The following figure shows an application-defined function, MyCompareObjects, that compares two OSA event objects of any object class supported by the application.

```
OSErr MyCompareObjects (DescType comparisonOperator, AEDesc theObject,
                        AEDesc objectOrDescToCompare, BOOL result)
{
  OSErr   myErr;
  result = FALSE;

  switch (comparisonOperator) {
      case kAEEquals:
```

```
        /*compare two objects for equivalence*/
        myErr = MyCompEquals(theObject, objectOrDescToCompare, result);
        return (myErr);
        break;
      case kAEGreaterThan:
        /*compare two objects for greater than*/
        myErr = MyCompGreaterThan(theObject, objectOrDescToCompare, result)
        return (myErr);
        break;
      case kAELessThan:
        /*compare two objects for less than*/
        myErr = MyCompLessThan(theObject, objectOrDescToCompare, result);
        return (myErr);
        break;
      default:
        /*this app does not support any other comparison operators*/
        return (errAEEventNotHandled);
        break;
  }
}
```

The MyCompareObjects function calls a separate application-defined routine for each comparison operator. In each case, the application-defined routine that actually performs the comparison can compare an OSA event object with either another OSA event object or with a descriptor record's data. If for any reason the comparison cannot be performed, the MyCompareObjects function returns the result code errAEEventNotHandled.

Writing Marking Callback Functions

Marking callback functions allow applications such as databases that can mark their own objects to take advantage of that capability when resolving object specifier records. Instead of returning a list of tokens for a group of OSA event objects that pass a test, your application can simply mark the OSA event objects and return a token that identifies how they have been marked. In this way, you can speed the resolution of complex object specifier records and reduce the amount of memory you need to allocate for tokens.

The use of marking callback functions is optional and usually makes sense if (a) you can reasonably expect that the tokens created in the process of resolving some object specifier records might not all fit in memory at once or (b) your application already uses a marking mechanism. If you want the OSA Event Manager to use marking callback functions provided by your application, you must add the kAEIDoMarking constant to the value of the *callbackFlags* parameter for the AEResolve function. If, for any reason, your application cannot mark a requested set of OSA event objects, it should return errAEEventNotHandled as the result code, and the OSA Event Manager will attempt to continue resolving the object specifier record by some other method, such as using a system marking function, if one exists.

If your application supports marking callback functions, it must provide three functions with declarations that match these examples:

```
OSErr MyGetMarkToken (AEDesc containerToken, DescType containerClass,
                      AEDesc Result)

OSErr MyMark (AEDesc theToken, AEDesc markToken, long markCount);

OSErr MyAdjustMarks (long newStart, long newStop, AEDesc markToken);
```

For more detailed information about these sample declarations, see Object Callback Functions.

To resolve a given object specifier record with the aid of the marking callback functions provided by your application, the OSA Event Manager first calls your application's *mark token function* (MyGetMarkToken), passing a token that identifies the container of the elements to be marked in the *containerToken* parameter and the container's object class in the *containerClass* parameter. The mark token function returns a mark token. A *mark token* , like other tokens, can be a descriptor record of any type; however, unlike other tokens, it identifies the way your application marks OSA event objects during the current session while resolving a single test. A mark token does not identify a specific OSA event object; rather, it allows your application to associate a group of objects with a marked set.

After it receives the mark token, the OSA Event Manager can call your application's *object-marking function* (MyMark) repeatedly to mark specific OSA event objects. The OSA Event Manager passes the following information to your marking function: in the *theToken* parameter, a token for the object to be marked (obtained from the appropriate object accessor function); in the *markToken* parameter, the current mark token; and in the *markCount* parameter, the mark count. The *mark count* indicates the number of times the OSA Event Manager has called the marking function for the current mark token. Your application should associate the mark count with each OSA event object it marks.

When the OSA Event Manager needs to identify either a range of elements or the absolute position of an element in a group of OSA event objects that pass a test, it can use your application's *mark-adjusting function* (MyAdjustMarks) to unmark objects that it has previously marked. For example, suppose an object specifier record specifies "any row in the table 'MyCustomers' for which the City column is 'Miami'". The OSA Event Manager first uses the appropriate object accessor routine to locate all the rows in the table for which the City column is "Miami" and calls the application's marking function repeatedly to mark them. It then generates a random number between 1 and the number of rows it found that passed the test and calls the application's mark-adjusting function to unmark all the rows whose mark count does not match the randomly generated number. If the randomly chosen row has a mark count value of 5, the OSA Event Manager passes the mark-adjusting function 5 in both the *newStart* parameter and the *newStop* parameter, and the current mark token in the *markToken* parameter. The *newStart* and *newStop* parameters identify the beginning and end of the new set of marked objects that the mark-adjusting function will create by unmarking those previously marked objects not included in the new set.

When the OSA Event Manager calls your mark-adjusting function, your application must dispose of any data structures that it may have created to mark the previously marked objects. The OSA Event Manager calls your mark-adjusting function only once for a given mark token.

A mark token is valid until the OSA Event Manager either disposes of it (by calling AEDisposeToken) or returns it as the result of the AEResolve function. If the final result of a call to the AEResolve function is a mark token, the OSA event objects currently marked for that mark token are those specified by the object specifier record passed to AEResolve, and your application can proceed to do whatever the OSA event has requested. Note that your application is responsible for disposing of a final mark token with a call to AEDisposeToken, just as for any other final token.

If your application supports marking, it should also provide a token disposal function. When the OSA Event Manager calls AEDisposeToken to dispose of a mark token that is not the final result of a call to AEResolve, the subsequent call to your token disposal function lets you know that you can unmark the OSA event objects marked with that mark token. A call to AEDisposeDesc to dispose of a mark token (which would occur if you did not provide a token disposal function) would leave the objects marked.

Creating Object Specifier Records

If your application creates and sends OSA events that require the target application to locate OSA event objects, your application must create object specifier records for those events. This section describes how to use the four keyword-specified descriptor records described in Descriptor Records Used in Object Specifier Records, to specify the object class ID, container, key form, and key data for an object specifier record.

Because the internal structure of an object specifier record is nearly identical to the internal structure of an AE record, it is possible to use AECreateList, AEPutPtr, and AEPutKeyDesc, to add the four keyword-specified descriptor records to an AE record, then use AECoerceDesc to coerce the AE record to a descriptor record of type typeObjectSpecifier. However, it is usually preferable to use the AECreateObjSpecifier function to accomplish the same goal. The AECreateObjSpecifier function adds the keyword-specified descriptor records directly to an object specifier record, thus eliminating several steps that are required if you create an AE record first. The instructions that follow make use of AECreateObjSpecifier.

To specify the class ID for an object specifier record, your application can specify the appropriate class ID value as the *desiredClass* parameter for the AECreateObjSpecifier function, which uses it to create a keyword-specified descriptor record with the keyword keyAEDesiredClass as part of an object specifier record.

To specify the container for an object specifier record, your application must create a keyword-specified descriptor record with the keyword keyAEContainer that fully describes the container of the OSA event object. Because this container is usually another OSA event object, the container is usually specified by another object specifier record.

To specify the complete container hierarchy of an OSA event object, your application must create a series of nested object specifier records, starting with the object specifier record for the OSA event object whose container is outermost. With the exception of this first object specifier record, each object specifier record specifies another object specifier record in the chain as a container.

For example, the figure in section Object Class shows a series of nested object specifier records that specify the first row of a table named "Summary of Sales" in a document named "Sales Report." The logical organization of the same object specifier records is summarized in the following table.

```
Keyword                 Descriptor type         Data

keyAEDesiredClass       typeType                cRow

keyAEContainer          typeObjectSpecifier     (see indented record)

  keyAEDesiredClass       typeType                cTable

  keyAEContainer          typeObjectSpecifier     (see indented record)
```

keyAEDesiredClass	typeType	cDocument
keyAEContainer	typeNull	Data handle is NULL
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeChar	"Sales Report"
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeChar	"Summary of Sales"
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	1

Note: The format used in the previous table and similar tables throughout this chapter does not show the structure of nested object specifier records as they exist within an OSA event. Instead, this format shows what you would obtain after calling AEGetKeyDesc repeatedly to extract the object specifier records from an OSA event record.

When you call AEGetKeyDesc to extract a null descriptor record, AEGetKeyDesc returns a descriptor record of type AEDesc with a descriptor type of typeNull and a data handle whose value is 0.

To specify the default container for an object specifier record (such as the container for the document in the previous table), you can use AECreateDesc to create a null descriptor record, which you can then pass in the *theContainer* parameter of the AECreateObjSpecifier function. The AECreateObjSpecifier function uses the null descriptor record to create a keyword-specified descriptor record with the keyword keyAEContainer as part of an object specifier record.

The object specifier record that specifies the default container is always the first record you create in a series of nested object specifier records that specifies the complete container hierarchy for an OSA event object. Each one in the series uses the previously created object specifier record to specify its container. As with the null descriptor record, you can pass an object specifier record as the *theContainer* parameter to the AECreateObjSpecifier function, which uses it to create a keyword-specified descriptor record with the keyword keyAEContainer.

To specify the key form for an object specifier record, your application can specify a key form constant as the *keyForm* parameter to the AECreateObjSpecifier function, which uses it to create a keyword-specified descriptor record with the keyword keyAEKeyForm as part of an object specifier record. The standard key forms for object specifier records are summarized in the table in section Key Data.

For example, the key form for the object specifier records in the previous table that specify the document and the table is formName. In other words, the key data identifies the document and the table by their names. Similarly, the key form for the object specifier record in the previous table that specifies the first row in the table is formAbsolutePosition. In other words, the key data identifies the position of the row compared to other rows in the same container.

To specify the key data for an object specifier record, your application must create a keyword-specified descriptor record with the keyword keyAEKeyData whose data handle refers to the appropriate data for the specified key form. You can use AECreateDesc, AECreateCompDescriptor, AECreateLogicalDescriptor, and related functions to create the descriptor record, which you can then pass in the *keyData* parameter of the AECreateObjSpecifier function. The AECreateObjSpecifier function uses this descriptor record to create a keyword-specified descriptor record with the keyword keyAEKeyData as part of an object specifier record.

Creating a Simple Object Specifier Record

This section shows how to use the AECreateObjSpecifier function to create the object specifier record shown in the table in section Creating Object Specifier Records. The AECreateObjSpecifier function creates the necessary keyword-specified descriptor records for the class ID, container, key form, and key data and returns the resulting object specifier record as a descriptor record of type typeObjectSpecifier.

The following code fragment shows how the AECreateObjSpecifier function creates an object specifier record from parameters that an application specifies.

```
DescType desiredClass;
AEDesc   myObjectContainer;
DescType myKeyForm;
AEDesc   myKeyDataDesc;
BOOL     disposeInputs;
AEDesc   myObjSpecRec;
OSErr     myErr;
```

```
desiredClass = cRow;
myObjectContainer = MyGetContainer;
myKeyForm = formAbsolutePosition;
myKeyDataDesc = MyGetKeyData;
disposeInputs = TRUE;

/*create an object specifier record*/
myErr = AECreateObjSpecifier(desiredClass, &myObjectContainer, myKeyForm,
                             &myKeyDataDesc, disposeInputs, &myObjSpecRec);
```

The code shown in the previous code fragment demonstrates how an application might use the AECreateObjSpecifier function to create four keyword-specified descriptor records as part of a descriptor record of type typeObjectSpecifier. The AECreateObjSpecifier function returns a result code of noErr if the object specifier record was successfully created. The object specifier record returned in the *myObjSpecRec* parameter describes an OSA event object of the class specified by the *desiredClass* parameter, located in the container specified by the *myObjectContainer* parameter, with the key form specified by the *myKeyForm* parameter and key data specified by the *myKeyDataDesc* parameter.

You can specify TRUE in the *disposeInputs* parameter if you want the AECreateObjSpecifier function to dispose of the descriptor records you created for the myObjectContainer and *myKeyDataDesc* parameters. If you specify FALSE, then your application is responsible for disposing of these leftover descriptor records.

The following code fragment shows an application-defined function that uses AECreateObjSpecifier to create an object specifier record for the first row in the table named "Summary of Sales" in the document "Sales Report," then uses the object specifier record returned in the *myObjSpecRec* parameter as the direct parameter for a Get Data event.

```
OSErr MyRequestRowFromTarget (AEAddressDesc targetAddress, OSAEvent reply)
{
  DescType desiredClass;
  DescType myKeyForm;
  AEDesc   myObjectContainer;
  AEDesc   myObjSpecRec;
  AEDesc   myKeyDataDesc;
  long     keyData;
  OSAEvent theOSAEvent;
  OSErr     myErr;
  OSErr     ignoreErr;

  /*initialize (set to null descriptor records) the two descriptor */
  /* records that must eventually be disposed of*/
  MyInit2DescRecs(myObjSpecRec, theOSAEvent);
  desiredClass = cRow; /*specify the class specify container for the row*/

  myErr = MyCreateTableContainer(myObjectContainer,
                    "Summary of Sales", "Sales Report");
  if (myErr == noErr) {
      myKeyForm = formAbsolutePosition;   /*specify the key form*/
      keyData = 1;                        /*specify the key data for row*/
      myErr = AECreateDesc(typeLongInteger, &keyData, sizeof(keyData),
                           &myKeyDataDesc);
      if (myErr == noErr)
        /*create the object specifier record*/
        myErr = AECreateObjSpecifier(desiredClass, myObjectContainer,
                                     myKeyForm, myKeyDataDesc, TRUE,
                                     myObjSpecRec);
      if (myErr == noErr)
        /*myObjSpecRec now describes an OSA event object, and will become *
        /* direct parameter of a Get Data event; first create Get Data */
        /* event*/
        myErr = AECreateOSAEvent(kAECoreSuite, kAEGetData, &targetAddress,
```

```
                                        kAutoGenerateReturnID, kAnyTransactionID,
                                        &theOSAEvent);

      if (myErr == noErr)
        /* add myObjSpecRec as the direct parameter of the Get */
        /* Data event */
        myErr = AEPutParamDesc(&theOSAEvent, keyDirectObject, &myObjSpecRec

      if (myErr == noErr)
        myErr = AESend(&theOSAEvent, &reply, kAEWaitReply + kAENeverInterac
                       kAENormalPriority, 120, NULL, NULL);
  }
  ignoreErr = AEDisposeDesc(&myObjSpecRec);
  ignoreErr = AEDisposeDesc(&theOSAEvent);
  return (myErr);
}
```

The MyRequestRowFromTarget function shown in the previous code fragment specifies the class ID as cRow, indicating that the desired OSA event object is a row in a table. It uses the application-defined function MyCreateTableContainer to create an object specifier record for the table that contains the row, passing "Summary of Sales" and "Sales Report" as the second and third parameters to identify the name of the table and the name of the document that contains the table. (The next section explains how to construct the MyCreateTableContainer function.) It then specifies the key form as the constant formAbsolutePosition, which indicates that the key data specifies the position of the row within its container; sets the *keyData* variable to 1, indicating the first row, and uses AECreateDesc to create a descriptor record for the key data; and uses AECreateObjSpecifier to create the object specifier record that describes the desired word.

The desired row is now fully described by the *myObjSpecRec* variable, which contains a descriptor record of type typeObjectSpecifier that contains the three nested object specifier records shown in the table in section Creating Object Specifier Records. After using AECreateOSAEvent to create a Get Data event, the MyRequestRowFromTarget function uses the AEPutParamDesc function to add the *myObjSpecRec* variable to the Get Data event as a direct parameter, then uses AESend to send the Get Data event.

Note that the MyRequestRowFromTarget function begins by using the application-defined function MyInit2DescRecs to set myObjSpecRec and theOSAEvent to null descriptor records. These two functions must be disposed of whether the function is successful or not. By setting them to null descriptor records, the function can dispose of them at the end regardless of where an error may have occurred.

Specifying the Container Hierarchy

Because the container for an object specifier record usually consists of a chain of other object specifier records that specify the container hierarchy, your application must create all the object specifier records in the chain, starting with the record for the outermost container. The following two code fragments demonstrate how to use the AECreateObjSpecifier function to create the first two object specifier records in such a chain: the records for a document and a table.

```
OSErr MyCreateDocContainer (AEDesc myDocContainer, char docName[NAMESIZE])
{
  AEDesc myDocDescRec;
  AEDesc nullDescRec;
  OSErr   myErr;

  /*create a descriptor record for the name of the document*/
  myErr = AECreateDesc(typeChar, &docName, strlen(docName), &myDocDescRec);
  if (myErr == noErr)     /*create a null descriptor record*/
      myErr = AECreateDesc(typeNull, NULL, 0, &nullDescRec);
  if (myErr == noErr)     /*reate an object specifier record to specify the
                          /* document object*/
      myErr = AECreateObjSpecifier(cDocument, nullDescRec, formName,
                                   myDocDescRec, TRUE, myDocContainer);
    return (myErr);
}
```

The function MyCreateDocContainer in The previous code fragment creates an object specifier record that identifies a document by name. It starts by using the AECreateDesc function to create two descriptor records: one of type typeChar for the name of the document, and one of type typeNull for the null descriptor record that specifies the default container (because the document is not contained in any other OSA event object). These two descriptor records can then be used as parameters for the AECreateObjSpecifier function, which returns an object specifier record (that is, a descriptor record of type typeObjectSpecifier) in the *myDocContainer* variable. The object specifier record specifies an OSA event object of the object class cDocument in the container specified by the *nullDescRec* variable with a key form of formName and the key data specified by the *myDocDescRec* variable. This object specifier can be used by itself to specify a document, or it can be used to specify the container for another OSA event object.

The following code fragment shows an application-defined function, MyCreateTableContainer, that creates an object specifier record describing a table contained in a document.

```
OSErr MyCreateTableContainer (AEDesc myTableContainer,
                              char tableName[NAMESIZE],
                              char docName[NAMESIZE])
{
  AEDesc myDocDescRec;
  AEDesc myTableDescRec;
  OSErr   myErr;

  /*create a container for the document*/
  myErr = MyCreateDocContainer(myDocDescRec, docName);
  if (myErr == noErr) {
      /*create the table container, first specify the descriptor record */
      /* for the key data*/
      myErr = AECreateDesc(typeChar, &tableName[1],
                           strlen(tableName), &myTableDescRec);
      if (myErr == noErr)
        myErr = AECreateObjSpecifier(cTable, myDocDescRec, formName,
                                     myTableDescRec, TRUE, myTableContaine
  }
  return (myErr);
}
```

The function MyCreateTableContainer in the previous code fragment starts by using the function MyCreateDocContainer from the second code fragment above to create an object specifier record that identifies the table's container-the document in which the table is located. Then it uses the AECreateDesc function to create a descriptor record for the key data-a name that, when combined with the key form formName, will identify the table in the document. The object specifier record for the document and the descriptor record specifying the table's name are passed to the function AECreateObjSpecifier. It returns an object specifier record in the *myTableContainer* parameter that specifies an OSA event object of the object class cTable in the container specified by the *MyDocDescRec* variable with a key form of formName and the key data specified by the *myTableDescRec* variable. This object specifier record can be used by itself to specify a table, or it can be used to specify the container for another OSA event object.

The second code fragment in section Creating Object Specifier Records uses the MyCreateTableContainer function shown in the previous code fragment to specify the container hierarchy illustrated in the table in section Creating Object Specifier Records. The nested object specifier records shown in that table use the key forms formName and formRelativePosition. You can create key data for the key forms formPropertyID, formUniqueID, and formRelativePosition using similar techniques.

Specifying a Property

The key form formPropertyID allows your application to specify key data identifying a property of the object specified as a container. For example, an object specifier record that identifies the font property of a word specifies cProperty as the class ID, an object specifier record for the word as the property's container, formPropertyID as the key form, and the constant pFont as the key data.

Note that an object specifier record that identifies a property does not include a value for the property, such as Palatino. The value of a property is returned or set as a parameter of an OSA event. For example, an application that sends a Get Data event to get the pFont property of a word receives a value such as Palatino in the *keyAEResult* parameter of the reply event, and an application that sends a Set Data event to change the pFont property of a word specifies a font in the *keyAEData* parameter of the Set Data event.

To specify the key data for a key form of formPropertyID, your application must create a descriptor record of typeType whose data consists of a constant specifying a property. You can use AECreateDesc to create a descriptor record that specifies the constant for a property, then use AECreateObjSpecifier to add the descriptor record to an object specifier record as a keyword-specified descriptor record with the keyword

keyAEKeyData.

For more information about object specifier records that specify a property, see Key Data for a Property ID.

Specifying a Relative Position

The key form formRelativePosition allows your application to specify key data identifying an element or a set of elements that are immediately before or after the specified container. For example, if the container is a table, you could use a key form of formRelativePosition to specify the paragraph before or after the table.

To specify the key data for a key form of formRelativePosition, your application must create a descriptor record of typeEnumerated whose data consists of a constant specifying either the element after (kAENext) or the element before (kAEPrevious) the specified container.

You can use AECreateDesc to create a descriptor record that specifies one of these constants, then use AECreateObjSpecifier to add it to an object specifier record as a keyword-specified descriptor record with the keyword keyAEKeyData.

For more information about object specifier records that specify a relative position, see Key Data for Relative Position.

Creating a Complex Object Specifier Record

This section describes how to create object specifier records that specify a test or a range. You can specify the object class ID for these object specifier records the same way you would for any other object specifier record. When you create the other three keyword-specified descriptor records, however, you can use additional OSA Event Manager routines and descriptor types to specify any combination of OSA event objects.

Specifying a Test

The key form formTest allows your application to specify key data that identifies one or more elements in the specified container that pass a test. To do so, your application must construct several interconnected descriptor records that specify comparisons and, if necessary, logical expressions.

For example, to specify "the first row in which the First Name column equals 'John' and the Last Name column equals 'Chapman' in the table 'MyAddresses' of the database 'WaveDB,'" your application must construct an object specifier record whose key data describes a logical expression that applies the logical operator AND to two separate comparisons for each row: a comparison of the First Name column to the word "John" and a comparison of the Last Name column to the word "Chapman."

The logical organization of the data for the object specifier record that specifies this test is summarized in the following two tables. (It is also illustrated in the figures in section Key Data for a Test). The listings in the remainder of this section demonstrate how to create this object specifier record. For general information about the organization of key data for a test, see Key Data for a Test.

The following table describes the object specifier record for the first row that meets a test in the table named "MyAddresses."

```
Keyword                 Descriptor type              Data
keyAEDesiredClass       typeType                     cRow
keyAEContainer          typeObjectSpecifier          (see indented record)
  keyAEDesiredClass       typeType                     cRow
  keyAEContainer          typeObjectSpecifier          (see indented record)
    keyAEDesiredClass       typeType                     cTable
    keyAEContainer          typeObjectSpecifier          (see indented record)
      keyAEDesiredClass       typeType                     cDatabase
      keyAEContainer          typeNull                     Data handle is NULL
      keyAEKeyForm            typeEnumerated               formName
```

keyAEKeyData	typeChar	"WaveDB"
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeChar	"MyAddresses"
keyAEKeyForm	typeEnumerated	formTest
keyAEKeyData	typeLogicalDescriptor	(see table in section Specifying a Test)
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	1

The following table describes the logical descriptor record which specifies a test.

Keyword	Descriptor type	Data
keyAELogicalOperator	typeEnumerated	kAEAnd
keyAELogicalTerms	typeAEList	(see indented records)
	typeCompDescriptor	(see indented record)
keyAECompOperator	typeType	kAEEquals
keyAEObject1	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cColumn
keyAEContainer	typeObjectBeingExamined	Data handle is NULL
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeChar	"First Name"
keyAEObject2	typeChar	"John"
typeCompDescriptor	(see indented record)	keyAECompOperator
typeType	kAEEquals	keyAEObject1
	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cColumn
keyAEContainer	typeObjectBeingExamined	Data handle is NULL
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeChar	"Last Name"
keyAEObject2	typeChar	"Chapman"

Because both the database and the table shown in The second table above are specified by name, it would be convenient to have an application-defined routine that creates an object specifier record that uses the key form formName. The MyCreateFormNameObjSpecifier function shown in the following code fragment can be used for this purpose.

```
OSErr MyCreateFormNameObjSpecifier (DescType class, AEDesc container,
                                    char keyDataName[NAMESIZE],
                                    AEDesc resultObjSpecRec)
{
  AEDesc keyDataDescRec;
  OSErr   myErr;

  myErr = AECreateDesc(typeChar, &keyDataName[1],
                       strlen(keyDataName), &keyDataDescRec);
  if (myErr == noErr)
      myErr = AECreateObjSpecifier(class, container, formName, keyDataDescR
                                   TRUE, resultObjSpecRec);
```

```
  return (myErr);
}
```

The MyCreateFormNameObjSpecifier function shown in the previous code fragment returns, in the *resultObjSpecRec* parameter, an object specifier record that describes an OSA event object of the class specified by the *class* parameter, located in the container specified by the *container* parameter, with the key form formName and key data specified by the *keyDataName* parameter. This function is used in the code three code fragments below to create object specifier records that use the key form formName for the database and the table.

The nested object specifier records shown in the previous table specify "the rows in which the First Name column equals 'John' and the Last Name column equals 'Chapman.'" To identify the rows that pass this test, the OSA Event Manager needs to evaluate two comparisons: the comparison of each row of the First Name column to the word "John," and the comparison of each row of the Last Name column to the word "Chapman."

The OSA Event Manager uses the information in comparison descriptor records to compare the specified elements in a container, one at a time, either to another OSA event object or to the data associated with a descriptor record. The two comparison descriptor records you need to create for this example are summarized in the previous table.

You can use the AECreateCompDescriptor function to create a comparison descriptor record, or you can create an AE record and use AECoerceDesc to coerce it to a comparison descriptor record. The following code fragment shows an example of an application-defined routine that creates an object specifier record and a descriptor record of typeChar, then uses the AECreateCompDescriptor function to add them to a comparison descriptor record.

```
OSErr MyCreateComparisonDescRec (AEDesc compDesc, char colName[NAMESIZE],
                                 char name[NAMESIZE])
{
  AEDesc logicalContainer, colNameDesc, nameDesc;
  AEDesc myObjectExaminedContainer;
  OSErr   myErr;

  /* create the object specifier record for keyAEObject1; */
  /* first create container */
  /* create key data */
  myErr = AECreateDesc(typeObjectBeingExamined, NULL, 0,
                       &myObjectExaminedContainer);
  if (myErr == noErr)
      /* now create the object specifier record */
      myErr = AECreateDesc(typeChar, &colName[1],
                           strlen(colName), &colNameDesc);
  if (myErr == noErr)
      myErr = AECreateObjSpecifier(cColumn, &myObjectExaminedContainer,
                                   &formName, colNameDesc, TRUE,
                                   &logicalContainer);
  if (myErr == noErr)
      /* create the descriptor record for keyAEObject2 */
      myErr = AECreateDesc(typeChar, &name[1], strlen(name), &nameDesc);
  if (myErr == noErr)
      /* create the first logical term (comp descriptor record) */
      myErr = AECreateCompDescriptor(kAEEquals, &logicalContainer,
                                     &nameDesc, TRUE, compDesc);
  return (myErr);
}
```

The MyCreateComparisonDescRec function takes two strings and uses them to create a comparison descriptor record. The string passed in the *colName* parameter specifies the name of the column whose contents should be compared to the string passed in the *name* parameter. First, the MyCreateComparisonDescRec function uses AECreateDesc to create a descriptor record of typeObjectBeingExamined, which is returned in the *myObjectExaminedContainer* variable. Next, AECreateDesc creates a descriptor record of descriptor type typeChar, whose data consists of the string in the variable *colName* , and which is returned in the variable *colNameDesc* . The code then passes the variables *myObjectExaminedContainer* and colNameDesc to the AECreateObjSpecifier function, which uses them to create an object specifier record, returned in the *logicalContainer* variable, that becomes the keyword-specified descriptor record with the keyword keyAEObject1.

Next, the MyCreateComparisonDescRec function uses AECreateDesc and the *name* parameter to create the descriptor record for keyAEObject2, which AECreateDesc returns in the *nameDesc* variable. Finally, the code passes the constant kAEEquals, the variable *logicalContainer* , and the variable nameDesc to the AECreateCompDescriptor function, which creates a comparison descriptor record that

allows the OSA Event Manager (with the help of object-comparison functions provided by the server application) to determine whether the specified column in the row currently being checked equals the specified string.

You can use the MyCreateComparisonDescRec function to create both the descriptor records of type typeCompDescriptor shown in the previous table. These descriptor records provide two logical terms for a logical descriptor record. The entire logical descriptor record corresponds to the logical expression "the First Name column equals 'John' AND the Last Name column equals 'Chapman.'"

You can use the AECreateLogicalDescriptor function to create a logical descriptor record, or you can create an AE record and use the AECoerceDesc function to coerce it to a comparison descriptor record. The following code fragment shows an application-defined function that adds two comparison descriptor records to a descriptor list, then uses the AECreateLogicalDescriptor function to create a logical descriptor record whose logical terms are the two comparison descriptor records.

```
OSErr MyCreateLogicalDescRec (AEDesc compDesc1, AEDesc compDesc2,
                              DescType logicalOperator, AEDesc logicalDesc)
{
  AEDescList logicalTermsList;
  OSErr myErr;

  /* create a logical descriptor record that contains two */
  /* comparison descriptor records */
  /* first create a list */
  myErr = AECreateList(NULL, 0, FALSE, &logicalTermsList);
  if (myErr == noErr)
        myErr = AEPutDesc(&logicalTermsList, 1, &compDesc1);
  if (myErr == noErr)
        myErr = AEPutDesc(&logicalTermsList, 2, &compDesc2);
  if (myErr == noErr)
        myErr = AEDisposeDesc(&compDesc1);
  if (myErr == noErr)
        myErr = AEDisposeDesc(&compDesc2);
  if (myErr == noErr)
        myErr = AECreateLogicalDescriptor(&logicalTermsList, logicalOperatc
                                          TRUE, &logicalDesc);
  return (myErr);
}
```

The following code fragment uses the application-defined functions shown in the previous three code fragment to build the object specifier record illustrated in the previous two tables.

```
OSErr MyCreateObjSpecRec (AEDesc theResultObj)
{
  AEDesc nullContainer;
  AEDesc databaseContainer;
  AEDesc tableContainer;
  AEDesc compDesc1;
  AEDesc compDesc2;
  AEDesc logicalTestDesc;
  AEDesc rowTestContainer;
  AEDesc rowOffset;
  OSErr myErr;

  /* create a null container */
  myErr = AECreateDesc(typeNull, NULL, 0, &nullContainer);
  /* create a container for the database */
  if (myErr == noErr)
      myErr = MyCreateFormNameObjSpecifier(cDatabase, nullContainer,
                                           "WaveDB", databaseContainer);
  if (myErr == noErr)
      /*create a container for the table*/
```

```
        myErr = MyCreateFormNameObjSpecifier(cTable, databaseContainer,
                                             "MyAddresses", tableContainer);
    /* create a container for the row--an object specifier record that */
    /* specifies a test (the row whose First Name column = "John" and */
    /* Last Name column = "Chapman") */
    /* create the first comparison descriptor record */
    if (myErr == noErr)
        myErr = MyCreateComparisonDescRec(compDesc1, "First Name", "John");
    /* create the second comparison descriptor record */
    if (myErr == noErr)
        myErr = MyCreateComparisonDescRec(compDesc2, "Last Name", "Chapman");
    /* create the logical descriptor record*/
    if (myErr == noErr)
        myErr = MyCreateLogicalDescRec(compDesc1, compDesc2, kAEAND,
                                       logicalTestDesc);
    /* now cr4ate the object specifier record that specifies the test */
    if (myErr == noErr)
        myErr = AECreateObjSpecifier(cRow, &tableContainer, &formTest,
                                     logicalTestDesc, TRUE, &rowTestContainer)
    /* create the object specifier record for the OSA event object */
    /* first, create the descriptor record for the key data */
    if (myErr == noErr)
        myErr = AECreateOffsetDescriptor (1, &rowOffset);
    /* now create the object specifier record */
    if (myErr == noErr)
        myErr = AECreateObjSpecifier(cRow, &rowTestContainer,
                                     &formAbsolutePosition, rowOffset,
                                     TRUE, &theResultObj);
    return (myErr);
}
```

The MyCreateObjSpecRec function shown in the previous code fragment begins by using AECreateDesc to create a null descriptor record, then uses the MyCreateFormNameObjSpecifier function (shown in the first code fragment in this section) to specify the default container for the database named "WaveDB." The code then calls the MyCreateFormNameObjSpecifier function again, this time passing the object specifier record for WaveDB to specify the container for the table "MyAddresses." The next two calls are both to the MyCreateComparisonDescRec function (shown in the second code fragment in this section), which creates the comparison descriptor records that allow the OSA Event Manager to compare the First Name column and Last Name column to the names "John" and "Chapman," respectively. The next call passes these two comparison descriptor records to the MyCreateLogicalDescRec function (shown in the third code fragment in this section) in the co*mpDesc1* and co*mpDesc2* variables.

Now all the components of the logical descriptor record are ready to assemble. The next call, to AECreateObjSpecifier, specifies the logical descriptor record in the *logicalTestDesc* variable as the key data for the object specifier record that specifies the test. A call to the OSA Event Manager routine AECreateOffsetDescriptor then creates an offset descriptor record that contains the integer 1. Finally, the code passes the offset descriptor record to the AECreateObjSpecifier function in the *rowOffset* variable to create the final object specifier record, which describes the requested row as the first row that passes the test.

The AECreateOffsetDescriptor function creates a descriptor record of type typeLongInteger that can be used as the key data with a key form of formAbsolutePosition to indicate an element's offset within its container. A positive integer indicates an offset from the beginning of the container (the first element has an offset of 1), and a negative integer indicates an offset from the end of the container (the last element has an offset of -1). Using AECreateOffsetDescriptor accomplishes the same thing as setting a variable to an integer and passing the variable to AECreateDesc to create a descriptor record of type typeLongInteger.

Specifying a Range

The key form formRange allows your application to specify key data that identifies a range of elements in the specified container. To do so, your application must first create a range descriptor record. The OSA Event Manager uses a range descriptor record to identify the two OSA event objects that specify the beginning and end of a range of elements.

For example, an object specifier record for a range of text in a document could specify the table named "Summary of Sales" as the first

boundary object and the figure named "Best-Selling Widgets for 1996" as the second boundary object for a range that consists of all the text between the table and the figure. Any word processor that keeps track of the relative positions of text, tables, and figures should be capable of supporting such a request.

The following table summarizes the logical organization of the data for the object specifier record that specifies this range. For general information about the organization of data within a range descriptor record, see Key Data for a Range.

Keyword	Descriptor type	Data
keyAERangeStart	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cTable
keyAEContainer	typeCurrentContainer	Data handle is NULL
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeChar	"Summary of Sales"
keyAERangeStop	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cFigure
keyAEContainer	typeCurrentContainer	Data handle is NULL
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeChar	"Best-Selling Widgets for 1996"

You can use the AECreateRangeDescriptor function to create a range descriptor record, or you can create an AE record and use AECoerceDesc to coerce it to a range descriptor record. The following code fragment provides an example of an application-defined routine that creates two object specifier records, then uses the AECreateRangeDescriptor function to add them to a range descriptor record.

The container for the boundary objects in the range descriptor record created by the following code fragment is the same as the container for the range itself. The object specifier record for the range's container is added to an object specifier record of key form formRange at the same time that the range descriptor record is added as key data. The container for the two boundary objects can therefore be specified in the range descriptor record by a descriptor record of type typeCurrentContainer whose data handle has the value NULL. The OSA Event Manager interprets this as a placeholder for the range's container when it is resolving the range descriptor record.

```
OSErr MyCreateRangeDescriptor (AEDesc rangeDescRec)
{
  AEDesc rangeStart;
  AEDesc rangeEnd;
  AEDesc currentContainer;
  AEDesc tableNameDescRec;
  AEDesc figureNameDescRec;
  OSErr   myErr;

  /* create the object specifier record for the start of the range */
  /* (the table named "Summary of Sales" in 'MyDoc' document) */

  /*create a descriptor record of type typeCurrentContainer*/
  myErr = AECreateDesc(typeCurrentContainer, NULL, 0, &currentContainer);

  /* create the object specifier record */
  if (myErr == noErr)
      myErr = MyCreateNameDescRec(tableNameDescRec, "Summary of Sales");
  if (myErr == noErr)
      myErr = AECreateObjSpecifier(cTable, &currentContainer, &formName,
                                   tableNameDescRec, FALSE, &rangeStart);

  myErr = AEDisposeDesc(&tableNameDescRec);
  /* create the object specifier record for the end of the range */
```

```
  /* (the figure named "Best-Selling Widgets..." in "MyDoc") */
  if (myErr == noErr)
      myErr = MyCreateNameDescRec(figureNameDescRec,
                                  "Best-Selling Widgets for 1996");

  if (myErr == noErr)
      myErr = AECreateObjSpecifier(cFigure, &currentContainer, &formName,
                                  figureNameDescRec, TRUE, &rangeEnd);

  /* now create the range descriptor record */
  if (myErr == noErr)
      myErr = AECreateRangeDescriptor(&rangeStart, &rangeEnd, TRUE, &rangeD

  return (myErr);
}
```

After creating a descriptor record of type typeCurrentContainer and a descriptor record for the first table's name, the previous code fragment uses the AECreateObjSpecifier function to create an object specifier record that identifies the beginning of the range. The parameters to AECreateObjSpecifier specify that the beginning of the range is an OSA event object of the object class cTable in the current container, with a key form of formName and key data that identifies the table by name. A second call to AECreateObjSpecifier creates the object specifier record that identifies the end of the range-an OSA event object of the cFigure object class in the current container, with a key form of formName and key data that identifies the figure by name. Finally, the code in the previous code fragment uses the AECreateRangeDescriptor function to create the range descriptor record, using the two previously created object specifier records to specify the beginning and end of the range.

Reference to Resolving and Creating Object Specifier Records

This section describes the OSA Event Manager routines your application can use to resolve and create object specifier records. It also describes application-defined object accessor functions and object callback functions that your application can provide for use by the OSA Event Manager in resolving object specifier records.

The first section, Data Structures Used in Object Specifier Records, summarizes the descriptor types and associated data that can be used in an object specifier record. Routines for Resolving and Creating Object Specifier Records describes the OSA Event Manager routines you use to initialize the Object Support Library, resolve object specifier records, set and manipulate object accessor functions, deallocate memory for tokens, and create object specifier records. Application-Defined Routines describes the object accessor functions and object callback functions that a server application can provide.

Data Structures Used in Object Specifier Records

The data for object specifier records can be specified using a variety of descriptor records and descriptor types. These are described in detail in Descriptor Records Used in Object Specifier Records and summarized in the following table.

Keyword	Descriptor type	Data
keyAEDesiredClass	typeType	Object class ID
keyAEContainer	typeObjectSpecifier	Object specifier record
	typeNull	Data handle is NULL. Specifies the default container at the top of the container hierarchy.
	typeObjectBeingExamined	Data handle is NULL. Specifies the container for elements that are tested one at a

		time; used only with formTest.
	typeCurrentContainer	Data handle is NULL. Specifies a container for an element that demarcates one boundary in a range. Used only with formRange.
keyAEKeyForm	typeEnumerated	formPropertyID formName formUniqueID formAbsolutePosition formRelativePosition formTest formRange formWhose
keyAEKeyData		(see indented key forms)
for formPropertyID	typeType	Property ID for an element's property
for formName	typeChar or other text type	Element's name
for formUniqueID	Any appropriate type	Element's unique ID
for formAbsolutePosition	typeLongInteger	Offset from beginning (positive) or end (negative) of container
	typeAbsoluteOrdinal	kAEFirst kAEMiddle kAELast kAEAny kAEAll
for formRelativePosition	typeEnumerated	kAENext kAEPrevious
for formTest	typeCompDescriptor typeLogicalDescriptor	(see tables in section Key Data for a Test)
for formRange	typeRangeDescriptor	(see table in section Key Data for a Range)
for formWhose	typeWhoseDescriptor	(see table in section Handling Whose Tests)

Routines for Resolving and Creating Object Specifier Records

This section describes routines for initializing the Object Support Library, resolving object specifier records, setting and manipulating object accessor functions, deallocating memory for tokens, and creating object specifier records.

Setting Object Accessor Functions and Object Callback Functions

The OSA Event Manager provides two routines that allow you to specify the object accessor functions and object callback functions provided by your application. The AEInstallObjectAccessor function adds an entry for an object accessor function to either the application's object accessor dispatch table or the system object accessor dispatch table. The AESetObjectCallbacks function allows you to specify the object

callback functions to be called for your application.

Getting, Calling, and Removing Object Accessor Functions

The OSA Event Manager provides three functions that allow you to get, call, and remove object accessor functions that you have installed in either the system or application object accessor dispatch table with the AEInstallObjectAccessor function. The AEGetObjectAccessor and AECallObjectAccessor functions get and call object accessor functions installed in the dispatch table you specify, and AERemoveObjectAccessor removes an installed function.

Resolving Object Specifier Records

If an OSA event parameter consists of an object specifier record, your handler for the event typically calls the AEResolve function to begin the process of resolving the object specifier record.

Deallocating Memory for Tokens

Whenever the AEResolve function returns a final token to your event handler as the result of the resolution of an object specifier record passed to AEResolve, your application can call the AEDisposeToken function to deallocate the memory used by the token.

Creating Object Specifier Records

The OSA Event Manager provides five functions that you can use to create some of the components of an object specifier record or to assemble an object specifier record:

- The AECreateOffsetDescriptor function creates an offset descriptor record, which specifies the position of an element in relation to the beginning or end of its container.
- The AECreateCompDescriptor function creates a comparison descriptor record, which specifies how to compare one or more OSA event objects with either another OSA event object or a descriptor record.
- The AECreateLogicalDescriptor function creates a logical descriptor record, which specifies a logical operator and one or more logical terms for the OSA Event Manager to evaluate.
- The AECreateRangeDescriptor function creates a range descriptor record, which specifies a series of consecutive elements in the same container.
- The AECreateObjSpecifier function assembles an object specifier record, which identifies one or more OSA event objects, from other descriptor records.

Instead of using these functions, you can create the corresponding descriptor records yourself using the AECreateDesc function, add them to an AE record using other OSA Event Manager routines, and coerce the AE record to a descriptor record of type typeObjectSpecifier. However, in most cases it is easier to use the functions listed in this section.

All of these functions except for AECreateOffsetDescriptor include a *disposeInputs* parameter. If the value of this parameter is TRUE, the function automatically disposes of any descriptor records you have provided as parameters to the function. If the value is FALSE, the application must dispose of the records itself. A value of FALSE may be more efficient for some applications because it allows them to reuse descriptor records.

For more information about these functions and examples of their use, see Creating Object Specifier Records.

Application-Defined Routines

The AEResolve function performs tasks that are required to resolve any object specifier record, such as parsing its contents, keeping track of the results of tests, and handling memory management. When necessary, AEResolve calls application-defined functions to perform tasks that are unique to the application, such as locating a specific OSA event object in the application's data structures or counting the number of OSA event objects in a container.

AEResolve can call two kinds of application-defined functions:

- *Object accessor functions* locate OSA event objects. Every application that supports simple object specifier records must provide one or more object accessor functions.
- *Object callback functions* perform other tasks that only an application can perform, such as counting, comparing, or marking OSA event objects. You can provide up to seven object callback functions, depending on the needs of your application.

This section provides model declarations for the object accessor functions and object callback functions that your application can provide.

Object Accessor Functions

You must provide one or more object accessor functions that can locate all the element classes and properties listed in the *OSA Event Registry: Standard Suites* for the object classes supported by your application. This section provides the routine declaration for an object accessor function.

MyObjectAccessor

Object accessor functions locate OSA event objects of a specified object class in a container identified by a token of a specified descriptor type.

Syntax

```
#define INCL_OSA

#include <os2.h>

OSErr MyObjectAccessor (DescType desiredClass, const AEDesc *containerToken
                        DescType containerClass, DescType keyForm,
                        const AEDesc *keyData, AEDesc *theToken,
                        long *theRefcon)
```

Parameters

desiredClass (DescType) - input
The object class of the desired OSA event objects.

containerToken (const AEDesc *) - input
A token that specifies the container of the desired OSA event objects.

containerClass (DescType) - input
The object class of the container.

keyForm (DescType) - input
The key form specified by the object specifier record being resolved.

keyData (const AEDesc *) - input
The key data specified by the object specifier record being resolved.

theToken (AEDesc *) - input
The token returned by the MyObjectAccessor function.

theRefcon (long *) - input
A reference constant that the OSA Event Manager passes to the object accessor function each time it is called.

Returns

rc (OSErr) - returns
Return code.

noErr	No error.
errAEEventNotHandled	The object-accessor function is unable to locate the requested OSA event objects.

Remarks

Each object accessor function provided by your application should either find elements of a specified object class or find properties of an OSA event object. The AEResolve function uses the object class ID of the specified OSA event object and the descriptor type of the token that identifies the object's container to determine which object accessor function to call. To install an object accessor function either in your application's object accessor dispatch table or in the system object accessor dispatch table, use the AEInstallObjectAccessor function.

Special Considerations

If the OSA Event Manager receives the result code errAEEventNotHandled after calling an object accessor function, it attempts to use other methods of locating the requested objects, such as calling an equivalent system object accessor function. Thus, an object accessor function that cannot locate a requested object should return errAEEventNotHandled. This allows the OSA Event Manager to try other object accessor functions that may be available.

Object Callback Functions

If an OSA event parameter consists of an object specifier record, your handler for the OSA event typically calls AEResolve to begin the process of locating the requested OSA event objects. The AEResolve function in turn calls object accessor functions and, if necessary, object callback functions provided by your application when it needs the information they can provide.

This section provides declarations for the seven object callback functions that your application can provide: the object-counting function (MyCountObjects), object-comparison function (MyCompareObjects), token disposal function (MyDisposeToken), error callback function (MyGetErrorDesc), mark token function (MyGetMarkToken), object-marking function (MyMark), and mark-adjusting function (MyAdjustMarks).

For information about writing and installing object callback functions, see Writing Object Callback Functions.

MyCountObjects

If you want the OSA Event Manager to help your application resolve object specifier records of key form formTest (and if your application does not specify kAEIDoWhose function) you should provide an object-counting function and an object-comparison function. An object-counting function counts the number of OSA event objects of a specified class in a specified container.

Syntax

```
#define INCL_OSA

#include <os2.h>

(OSErr MyCountObjects (DescType desiredClass, DescType containerClass,
                       const AEDesc *theContainer, long *result)
```

Parameters

desiredClass (DescType) - input
The object class of the OSA event objects to be counted.

containerClass (DescType) - input
The object class of the container for the OSA event objects to be counted.

theContainer (const AEDesc *) - input
A token that identifies the container for the OSA event objects to be counted.

result (long *) - input
Your object-counting function should return, in this parameter, the number of OSA event objects of the specified class in the specified container.

Returns

rc (OSErr) - returns
Return code.

noErr	No error.
errAEEventNotHandled	The object-counting function is unable to count the specified OSA event objects.

Remarks

The OSA Event Manager calls your object-counting function when, in the course of resolving an object specifier record, the manager requires a count of the number of OSA event objects of a given class in a given container.

Special Considerations

If the OSA Event Manager receives the result code errAEEventNotHandled after calling an object-counting function, it attempts to use other methods of counting the specified objects, such as calling an equivalent system object-counting function. Thus, an object-counting function that cannot count the specified objects should return errAEEventNotHandled. This allows the OSA Event Manager to try other object-counting functions that may be available.

MyCompareObjects

If you want the OSA Event Manager to help your application resolve object specifier records of key form formTest (and if your application does not specify kAEIDoWhose function), you should provide an object-counting function and an object-comparison function. After comparing one OSA event object to another or to the data for a descriptor record, an object-comparison function should return TRUE or FALSE in the *result* parameter.

Syntax

```
#define INCL_OSA

#include <os2.h>

OSErr MyCompareObjects (DescType comparisonOperator, const AEDesc *theObjec
                        BOOL *result)
```

Parameters

comparisonOperator (DescType) - input
The comparison operator.

See the AECreateCompDescriptor function for standard comparison operators at the time of publication of this book. The current version of the *OSA Event Registry: Standard Suites* lists all the constants for comparison operators.

theObject (const AEDesc *) - input
A token.

objectOrDescToCompare (AEDesc *) - input
A token or some other descriptor record that specifies either an OSA event object or a value to compare to the OSA event object specified by the *object* parameter.

result (BOOL *) - input
Your object-comparison function should return, in this parameter, a Boolean value that indicates whether the values of the *object* and *objectOrDescToCompare* parameters have the relationship specified by the *comparisonOperator* parameter (TRUE) or not (FALSE).

Returns

rc (OSErr) - returns
Return code.

noErr	No error.
errAEEventNotHandled	The object-comparison function is unable to compare the specified OSA event objects.

Remarks

The OSA Event Manager calls your object-comparison function when, in the course of resolving an object specifier record, the manager needs to compare an OSA event object with another or with a value.

It is up to your application to interpret the comparison operators it receives. The meaning of comparison operators differs according to the OSA event objects being compared, and not all comparison operators apply to all object classes.

Special Considerations

If the OSA Event Manager receives the result code errAEEventNotHandled after calling an object-comparison function, it attempts to use other methods of comparison, such as calling an equivalent system object-comparison function. Thus, an object-comparison function that cannot perform a requested comparison should returnerrAEEventNotHandled. This allows the OSA Event Manager to try other object-comparison functions that may be available.

MyDisposeToken

If your application requires more than a call to the AEDisposeDesc function to dispose of a token, or if it supports marking callback functions, you must provide one token disposal function. A token disposal function disposes of a specified token.

Syntax

```
#define INCL_OSA

#include <os2.h>

OSErr MyDisposeToken (AEDesc *unneededToken)
```

Parameters

unneededToken (AEDesc *) - input
The token to dispose of.

Returns

rc (OSErr) - returns
Return code.

noErr	No error.
errAEEventNotHandled	The object-comparison function is unable to compare the specified OSA event objects.

Remarks

The OSA Event Manager calls your token disposal function whenever it needs to dispose of a token. It also calls your disposal function when your application calls the AEDisposeToken function. If your application does not provide a token disposal function, the OSA Event Manager calls AEDisposeDesc instead.

Your token disposal function must be able to dispose of all of the token types used by your application.

If your application supports marking, a call to MyDisposeToken to dispose of a mark token lets your application know that it can unmark the objects marked with that mark token.

Special Considerations

If the OSA Event Manager receives the result code errAEEventNotHandled after calling a token disposal function, it attempts to dispose of the token by some other method, such as calling an equivalent system token disposal function if one is available or, if that fails, by calling AEDisposeDesc. Thus, a token disposal function that cannot dispose of a token should return errAEEventNotHandled. This allows the OSA Event Manager to try other token disposal functions that may be available.

MyGetErrorDesc

If you want to find out which descriptor record is responsible for an error that occurs during a call to the AEResolve function, you can provide an error callback function. An error callback function returns a pointer to an address. The OSA Event Manager uses this address to store the descriptor record it is currently working with if an error occurs during a call to AEResolve.
Syntax

```
#define INCL_OSA

#include <os2.h>

OSErr MyGetErrorDesc (DescPtr *DescPtr)
```

Parameters

errDescPtr (DescPtr *) - input
A pointer to an address.

Returns

rc (OSErr) - returns
Return code.

noErr	No error.

Remarks

Your error callback function simply returns an address. Shortly after your application calls AEResolve, the OSA Event Manager calls your error callback function and writes a null descriptor record to the address returned, overwriting whatever was there previously. If an error occurs during the resolution of the object specifier record, the OSA Event Manager calls your error callback function again and writes the descriptor record-often an object specifier record-to the address returned. If AEResolve returns an error during the resolution of an object specifier record, this address contains the descriptor record responsible for the error.

Normally you should maintain a single global variable of type AEDesc whose address your error callback function returns no matter how many times it is called. Be careful if you use any other method. When recovering from an error, the OSA Event Manager never writes to the address you provide unless it already contains a null descriptor record. Thus, if you do not maintain a single global variable as just described, you should write null descriptor records to any addresses passed by your error callback function that are different from the addresses returned the first time your function is called after a given call to AEResolve.

If the result code returned by the MyGetErrorDesc function has a nonzero value, the OSA Event Manager continues to resolve the object specifier record as if it had never called the error callback function.

MyGetMarkToken

If your application supports marking, you must provide one mark token function. A mark token function returns a mark token.
Syntax

```
#define INCL_OSA


#include <os2.h>

OSErr MyGetMarkToken (const AEDesc *containerToken, DescType containerClass
                      AEDesc *result)
```

Parameters

containerToken (const AEDesc *) - input
The OSA event object that contains the elements to be marked with the mark token.

containerClass (DescType) - input
The object class of the container that contains the objects to be marked.

result (AEDesc *) - input
Your mark token function should return a mark token in this parameter.

Returns

rc (OSErr) - returns
Return code.

noErr	No error.
errAEEventNotHandled	The mark token function is unable to return a mark token; if the OSA Event Manager gets this result, it attempts to get a mark token by calling the equivalent system marking callback function.

Remarks

To get a mark token, the OSA Event Manager calls your mark token function. Like other tokens, the mark token returned can be a descriptor record of any type; however, unlike other tokens, a mark token identifies the way your application will mark OSA event objects during the current session while resolving a single object specifier record that specifies the key form formTest.

A mark token is valid until the OSA Event Manager either disposes of it (by calling AEDisposeToken) or returns it as the result of the AEResolve function. If the final result of a call to AEResolve is a mark token, the OSA event objects currently marked for that mark token are those specified by the object specifier record passed to AEResolve, and your application can proceed to do whatever the OSA event has requested. Note that your application is responsible for disposing of a final mark token with a call to AEDisposeToken, just as for any other final token.

If your application supports marking, it should also provide a token disposal function modeled after the token disposal function. When the OSA Event Manager calls AEDisposeToken to dispose of a mark token that is not the final result of a call to AEResolve, the subsequent call to your token disposal function lets you know that you can unmark the OSA event objects marked with that mark token. A call to AEDisposeDesc to dispose of a mark token (which would occur if you did not provide a token disposal function) would go unnoticed.

MyMark

If your application supports marking, you must provide one object-marking function. An object-marking function marks a specific OSA event object.
Syntax

```
#define INCL_OSA


#include <os2.h>

OSErr MyMark (const AEDesc *theToken, const AEDesc *markToken,
              long *markCount)
```

Parameters

theToken (const AEDesc *) - input
The token for the OSA event object to be marked.

markToken (const AEDesc *) - input
The mark token used to mark the OSA event object.

markCount (long *) - input
The number of times this function has been called for the current mark token (that is, the number of OSA event objects that have so far passed the test, including the element to be marked).

Returns

rc (OSErr) - returns
Return code.

noErr	No error.
errAEEventNotHandled	This function is unable to mark the specified OSA event object; if the OSA Event Manager gets this result, it attempts to mark the object by calling the equivalent system object-marking function.

Remarks

To mark an OSA event object using the current mark token, the OSA Event Manager calls the object-marking function provided by your application. In addition to marking the specified object, your MyMark function should record the mark count for each object that it marks. The mark count recorded for each marked object allows your application to determine which of a set of marked tokens pass a test, as described in the next section for the MyAdjustMarks function.

MyAdjustMarks

If your application supports marking, you must provide one mark-adjusting function. A mark-adjusting function adjusts the marks made with the current mark token.
Syntax

```
#define INCL_OSA

#include <os2.h>

OSErr MyAdjustMarks (long newStart, long newStop, const AEDesc *markToken)
```

Parameters

newStart (long) - input
The mark count value (provided when the MyMark callback routine was called to mark the object) for the first object in the new set of marked objects.

newStop (long) - input
The mark count value (provided when the MyMark callback routine was called to mark the object) for the last object in the new set of marked objects.

markToken (const AEDesc *) - input
The mark token for the marked objects.

Returns

rc (OSErr) - returns
Return code.

noErr	No error.
errAEEventNotHandled	The MyMark function is unable to mark the specified OSA event object; if the OSA Event Manager gets this result, it attempts to mark the object by calling the equivalent system object-marking function.

Remarks

When the OSA Event Manager needs to identify either a range of elements or the absolute position of an element in a group of OSA event objects that pass a test, it can use your application's mark-adjusting function to unmark objects previously marked by a call to your marking function. For example, suppose an object specifier record specifies "any row in the table 'MyCustomers' for which the City column is 'Miami.'" The OSA Event Manager first uses the appropriate object accessor function to locate all the rows in the table for which the City column is "Miami" and calls the application's marking function repeatedly to mark them. It then generates a random number between 1 and the number of rows it found that passed the test and calls the application's mark-adjusting function to unmark all the rows whose mark count does not match the randomly generated number. If the randomly chosen row has a mark count value of 5, the OSA Event Manager passes the value 5 to the mark-adjusting function in both the *newStart* parameter and the *newStop* parameter, and passes the current mark token in the *markToken* parameter. When the OSA Event Manager calls your MyAdjustMarks function, your application must dispose of any data structures that it created to mark the previously marked objects.

Summary of Resolving and Creating Object Specifier Records

The following table summarizes the methods used to resolve and create object specifier records.

Function Name	Description
AECallObjectAccessor	Invokes one of your application's object accessor functions.
AECreateCompDescriptor	Creates a comparison descriptor record.
AECreateLogicalDescriptor	Creates a logical descriptor record.
AECreateObjSpecifier	Creates an object specifier record.
AECreateOffsetDescriptor	Creates an offset descriptor record.
AECreateRangeDescriptor	Creates a range descriptor record.
AEDisposeToken	Deallocates the memory used by a token.
AEGetObjectAccessor	Gets a pointer to an object accessor function and the value of its reference constant.
AEInstallObjectAccessor	Adds an entry for an object accessor function to the application's object accessor dispatch table.
AERemoveObjectAccessor	Removes an object accessor function from the application's object accessor dispatch table.
AEResolve	Resolves an object specifier record in an OSA event parameter.
AESetObjectCallbacks	Specifies the object callback functions to be called for the application.

Introduction to Scripting

Many of the examples shown in this chapter are in the AppleScript language. AppleScript is not currently available on OS/2. Examples are also shown in the Object REXX language.

This chapter provides an overview of the tasks involved in making your application scriptable and recordable. This chapter also introduces some of the ways your application can use the Component Manager and scripting components to manipulate and execute scripts. The three chapters that follow provide detailed information, including sample code, about the topics introduced in this chapter.

Interapplication Communication describes the Open Scripting Architecture (OSA) and its relationship to the OSA Event Manager and other parts of the IAC architecture. If your application supports the appropriate core and functional-area events defined in the *OSA Event Registry: Standard Suite*s , you can make it scriptable (that is, capable of responding to OSA events sent by scripting components) by providing an OSA event terminology extension (**aete**) resource. This chapter describes some of the tasks involved in making your application scriptable and introduces the **aete** resource.

This chapter also introduces OSA event recording and the use of the standard scripting component routines to manipulate and execute scripts. Recording OSA Events describes in detail how to make your application recordable, and Scripting Components describes how to use the standard scripting component routines.

To use this chapter or any of the chapters that follow, you should be familiar with OSA Events and Responding to OSA Events. If you plan to make your application recordable, you should also read Creating and Sending OSA Events and Resolving and Creating Object Specifier Records.

If you are developing a scripting component, you should provide support for the standard scripting component routines described in ComponentManager.

This chapter begins with an overview of scripts and scripting components. The rest of the chapter describes how the OSA makes it possible to do the following:

- Make your application scriptable
- Make your application recordable
- Have your application manipulate and execute scripts

About Scripts and Scripting Components

A *script* is any collection of data that, when executed by the appropriate program, causes a corresponding action or series of actions. The Open Scripting Architecture (OSA) provides a standard mechanism that allows users to control multiple applications with scripts written in a variety of scripting languages. Each scripting language has a corresponding *scripting component* . Each scripting component supports the standard scripting component routines described in Scripting Components.

When a scripting component executes a script, it performs the actions described in the script, including sending OSA events to applications if necessary. Like other components that use the Component Manager, scripting components can provide their own routines in addition to the standard routines that must be supported by all components of the same type.

Scripting components typically implement a text-based scripting language based on OSA events. For example, the *Object REXX component* implements *Object REXX* , a user scripting language provided by IBM as part of OS/2.

Other scripting components may support the standard scripting component routines in different ways. Scripting components need not implement a text-based scripting language, or even one that is based on OSA events. For example, specialized scripting components can play sounds, execute applications, or perform almost any other action when they execute scripts.

This chapter describes three ways that you can take advantage of the OSA:

- You can make your application *scriptable* , or capable of responding to OSA events sent to it by a scripting component. An application is scriptable if it
    - Responds to the appropriate standard OSA events as described in Responding to OSA Events.
    - Provides an OSA event terminology extension (**aete**) resource describing the OSA events that your application supports and the user terminology that corresponds to those events. The **aete** resource allows scripting components to interpret scripts correctly and send the appropriate OSA events to your application during script execution.
- You can make your application *recordable* -that is, capable of sending OSA events to itself to report user actions to the OSA Event Manager for recording purposes. After a user has turned on recording for a particular scripting component, the scripting component receives a copy of every subsequent OSA event that any application on the local computer sends to itself. The scripting component records such events in the form of a script.
- You can have your application manipulate and execute scripts with the aid of a scripting component. To do so, your application must:
    - Use the Component Manager to open a connection with the appropriate component.
    - Use the standard scripting component routines described in Scripting Components to record, edit, compile, save, load, or execute scripts.

Users of scriptable applications can execute scripts to perform tasks that might otherwise be difficult to accomplish, especially repetitive or conditional tasks that involve multiple applications. For example, a user can execute an Object REXX script to locate database records with specific characteristics, create a series of graphs based on those records, import the graphs into a page-layout document, and send the document to a remote computer on the network via electronic mail. When a user executes such a script, the scripting component attempts to perform the actions the script describes, including sending OSA events when necessary.

To respond appropriately to the OSA events sent to it by the scripting component, the database application in this example must be able to locate records with specific characteristics so that it can identify and return the requested data. These characteristics are described by an object specifier record that is part of an OSA event supported by the application. Also, the other applications involved must support OSA events that manipulate the data in the ways described in the script. Each application in this example must also provide an **aete** resource describing the OSA events that the application supports and the user terminology that corresponds to those events, so that the scripting component can interpret the script correctly.

Even with little or no knowledge of a particular scripting language, users of applications that are recordable as well as scriptable can record simple scripts. More knowledgeable users may also wish to record their actions as scripts with recordable applications and then edit or combine scripts as needed.

An application that uses scripting components to manipulate and execute scripts need not be scriptable; however, if it is scriptable, it can execute scripts that control its own behavior. In other words, it can perform tasks by means of scripts and allow users to modify those scripts to suit their own needs.

The next three sections provide an overview of the way scripting components can interact with applications.

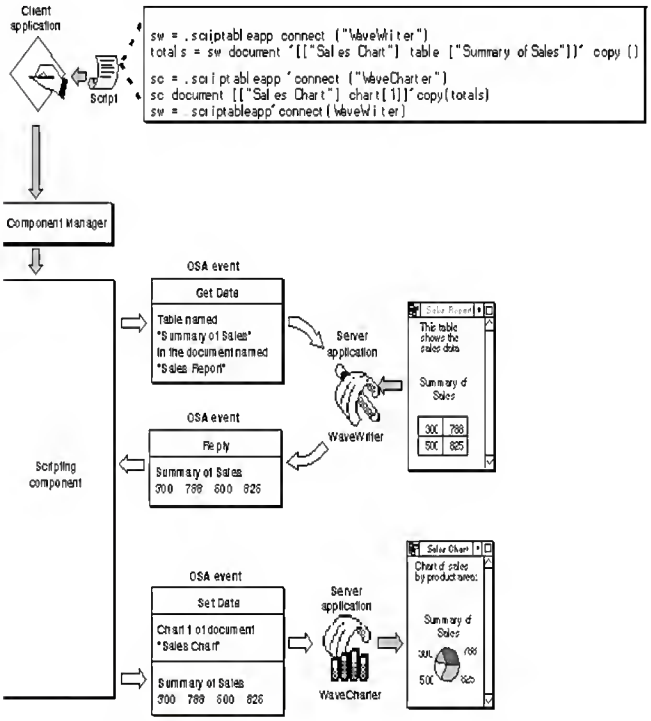
Script Editors and Script Files

A *script editor* is an application that allows users to record, edit, save, and execute scripts. For example, the Object REXX component uses the services of the Script Editor application.

Users with some knowledge of a text based scripting language, such as Object REXX, can use the Script Editor to modify recorded scripts or write their own scripts. The Script Editor provides entry-level scripting capabilities, but it is not intended for intensive script development. Users who wish to write complex scripts may replace Script Editor with more sophisticated editors that provide specialized debugging and development tools.

Scripting Components and Scriptable Applications

Scripting components control the behavior of scriptable applications by means of OSA events. For example, when the Object REXX component executes the scripting script it sends the OSA events shown in the following figure to trigger the actions described by the script. The client application in this example would most commonly be a script editor but could also be any other application that uses standard scripting component routines to manipulate and execute scripts.



As described in OSA Events, a client application is any application that uses OSA events to request a service or information. A client application that executes a script does not send the corresponding OSA events itself; instead, it uses scripting component routines to manipulate and execute the script. The scripting component sends OSA events when necessary to trigger the actions described in the script. Similarly, a scriptable application that responds to the OSA events sent by a scripting component can be considered the server application for those OSA events.

When a scripting component evaluates a script, it attempts to perform all the actions described in the script, including sending OSA events when necessary. In the example shown in the previous figure, the scripting component first performs the action described in the first tell statement:

```
sw = .scriptableapp~connect("WaveWriter")
totals = sw document["Sales Chart"] table["Summary of Sales"]~copy()
```

To perform this action, the Object REXX component sends a Get Data event to the WaveWriter application requesting the data from the specified table. The WaveWriter application returns the data to the Object REXX component in a standard reply OSA event, and the Object REXX component sets the value of the variable Totals to the data returned by WaveWriter.

Then the Object REXX component performs the action described in the second tell statement:

```
sc = .scriptableapp~connect("WaveCharter")
sc [document["Sales Chart"] chart[1]]~copy(totals))
```

In this case, the Object REXX component sends a Set Data event to the WaveCharter application that sets the specified chart to the value of the variable Totals.

Both WaveWriter and WaveCharter are server applications for the OSA events sent by the Object REXX component, because they are performing services in response to requests made by the client application via the script.

To send the appropriate OSA events to a scriptable application while executing a script, a scripting component must obtain information about the nature of that application's support for OSA events and the human-language terminology to associate with those events. A scriptable application provides this information in the form of an OSA event terminology extension (**aete**) resource. A scripting component uses both the **aete** resource provided by a scriptable application and the OSA event user terminology (**aeut**) resource provided by the scripting component itself to obtain the information it needs to execute a script that controls that application.

See Making Your Application Scriptable for an overview of the tasks you should perform to make your application scriptable and a more detailed description of the **aete** and **aeut** resources. See Making Your Application Recordable for an overview of the tasks you should perform if you want your application to be recordable as well as scriptable.

Scripting Components and Applications That Execute Scripts

To store and execute scripts as a client application, your application must first establish a connection with a scripting component registered with the Component Manager on the same computer. Each scripting component can manipulate and execute scripts written in the corresponding scripting language when your application calls the standard scripting component routines.

Your application can use scripting component routines to:

- Obtain a handle to a script in a form that can be saved, and load the script again when necessary
- Allow users to modify scripts that have been previously saved
- Compile and execute scripts
- Redirect OSA events to script contexts
- Supply application-defined functions for use by scripting components
- Control the recording process directly, turning recording off and on and saving the recorded script for use by your application

Note: The Object REXX scripting component does not currently support script contexts, event handling, or recording.

Your application can perform these tasks as a client application regardless of whether it is scriptable or recordable. If your application is scriptable, however, it can execute scripts that control its own behavior, thus acting as both the client application and the server application for the corresponding OSA events. For example, your application can allow users to associate a script with a custom menu command that performs a series of routine actions on a selected object, sets preferences, or automates other actions within your application.

You can also use scripting component routines to execute scripts that perform tasks for your application with the aid of other applications. For example, a user of a word-processing application might be able to attach a script to a specific word so that the application executes the script whenever that word is double-clicked. Such a script could trigger OSA events that cause other applications to look up and display related information, play a movie, perform a calculation, play a voice annotation, and so on.

Your application can associate a script with either OSA event objects or application-defined objects. Almost any user action can be used to trigger such a script: choosing a menu command, clicking a button, tabbing from one table cell to another, and so on. The script can be executed directly by the application when it detects a triggering action; or, if the script is associated with an OSA event object in the form of a script context, it can be executed automatically when a specified OSA event performs an action on that object.

The rest of this section describes one way that an application could execute such a script. Suppose a forms application allows users to create custom forms that can include scripts associated with specific fields on the form. These scripts are executed when the user presses **Enter** or **Tab** in the appropriate field. For the purposes of this example, it does not matter whether a field with which a script is associated is an OSA

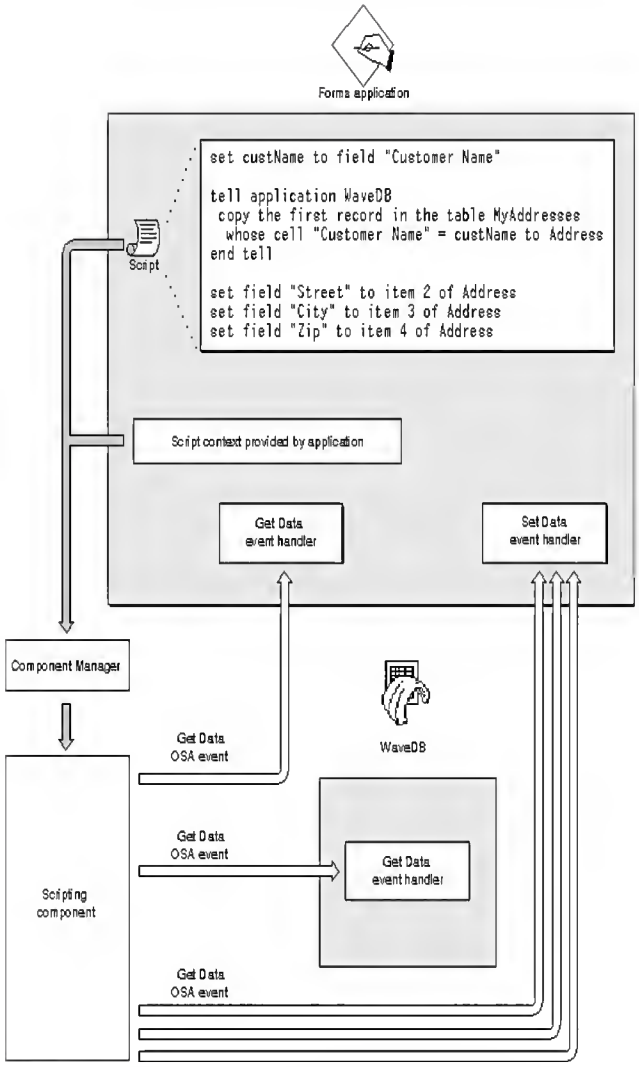
event object (which can be described in an object specifier record) or some other application-defined object (which cannot be described in an object specifier record).

A company could use the forms application to create a custom order form for taking telephone orders. If the customer has ordered from the company before, the user can quickly retrieve the customer's address from the company database by typing the customer's name in a field and pressing the **Tab** key. In response, the application executes the script associated with the field. The script might look like this:

```
set custName to field "Customer Name"

tell application WaveDB
    copy the first record in the table MyAddresses
          whose cell "Customer Name" = custName to Address
end tell

set field "Street" to item 2 of Address
set field "City" to item 3 of Address
set field "Zip" to item 4 of Address
```

To execute such a script (or to manipulate it any other way, such as when the form is first created), the forms application must previously have established a connection with the appropriate scripting component. When the user enters a customer name and presses **Tab**, the forms application calls scripting component routines to execute the script. As shown in the following figure, the scripting component first sends the forms application a Get Data event that requests the contents of the "Customer Name" field and sets the variable custName to that value. It then sends WaveDB a Get Data event that requests the appropriate address information and copies it to the variable Address. (The replies to the Get Data events are not shown in the following figure). Finally, the scripting component sends the forms application a Set Data event that copies the address information from the variable Address to the appropriate fields.

The scripting component needs to maintain the binding of the variables custName and Address throughout execution of the script. Scripting components can bind variables with the aid of a *script context* , which is a script that maintains context information for the execution of other scripts. An application specifies a script context when it executes a script. The forms application in the following figure provides a context for the scripting component to use whenever it executes a script associated with a button.



In the example shown in the previous figure, the application executes the script directly when the cursor is in the appropriate field and the user presses **Tab** or **Enter**. Your application can also associate such a script with an object in the form of a script context, so that the script context is executed whenever a specified OSA event acts on the field. Using a Script Context to Handle an OSA Event describes this approach in more detail.

See Manipulating and Executing Scripts for an overview of methods your application can use to save and load script data, compile source data, and perform other useful tasks with scripting component routines. Scripting Components provides full implementation details, including sample code and human interface guidelines for associating scripts with objects.

Making Your Application Scriptable

To make your application scriptable, you need to:

- Define a hierarchy of OSA event objects within your application that you want client applications to be able to identify-that is, which objects can be contained by other OSA event objects in your application, which properties each kind of object can have, and so on
- Write OSA event handlers, object accessor functions, and other routines required to implement the OSA events and related object classes that you want to support
- Create an **aete** resource

OSA Events, Responding to OSA Events, and Resolving and Creating Object Specifier Records describe how to perform the first two tasks. The extent to which scripts can control your application depends mainly on the extent of your application's support for OSA events. For example, if your application does not provide the OSA event handlers and object accessor functions required to locate and manipulate windows, users will not be able to use scripts to control your application's windows. Although you should use the definitions in the *OSA Event Registry: Standard Suites* whenever possible, you have considerable freedom to extend or limit your implementation of the standard OSA events according to the needs of your application.

The OSA makes it possible to design new kinds of applications that always operate in the background and can be controlled only by means of scripts. For example, it is possible to design a simple telecommunications program that can log on to a network, send and receive text files created by another application, and perform other basic operations in response to scripts without providing any other form of user interface. Such an application would not need to support OSA events that control window movement or menus; instead, it would need to support only those OSA events that execute its basic telecommunications operations.

At the other extreme, some applications allow users to arrange windows, palettes, and dialog boxes on their screen in many different ways, or to customize menus or other aspects of the presentation of information. If such an application can respond to scripts that control windows, dialog boxes, specialized preferences, and other aspects of the presentation of information, it can allow users who might not otherwise explore those capabilities to take advantage of them. For example, a naive user could execute a script that sets up a powerful word processor with the appropriate menus, window and palette arrangement, and formatting templates for a particular task, such as producing a company newsletter.

Scripting components use **aeut** and **aete** resources to associate OSA event codes supported by your application with corresponding human-language terms used in scripts that control your application. Each scripting component supplies an **aeut** resource, and each scriptable application provides an **aete** resource. The next section introduces the **aeut** and **aete** resources.

About OSA Event Terminology Resources

As explained in OSA Events, applications can support different combinations of the standard suites of OSA events. Applications can also extend the definitions of individual OSA events and object classes, or define custom OSA events and object classes. Scripting components use the *OSA event user terminology resources* , **aeut** and **aete**, to associate the IDs, keywords, and other codes used in OSA events with the corresponding human-language terms used in scripts that control your application.

The OSA event user terminology (**aeut**) resource contains terminology information for all the standard suites of OSA events defined in the *OSA Event Registry: Standard Suites* . The resource consists of a sequence of concatenated arrays that map human-language names to each of the following:

- ID defined for each suite
- OSA events defined for each suite
- Parameters defined for each OSA event
- OSA event object classes defined for each suite
- Properties defined for each object class
- Elements defined for each object class
- Key forms defined for each element class
- Comparison operators defined for each suite
- Values for enumerators defined for each suite

An **aete** resource has the same format as the **aeut** resource but serves a different purpose. Each scriptable application must include its own **aete** resource describing which of the standard suites listed in the **aeut** resource it supports and other application-specific information. Since the human-language equivalents for the standard suites are defined in the **aeut** resource, applications that support standard suites without any modifications do not have to define such equivalents; instead, they can simply list, in the **aete** resource, the suites they support. The scripting component associates the standard suites listed in the **aete** resource with the corresponding OSA event descriptions in its **aeut** resource.

Applications can also use the **aete** resource to describe extensions to the standard suites, such as additional parameters for standard OSA events, additional properties and element classes for the standard OSA event object classes, and additional key forms for each element class. Information about such extensions must be included in the appropriate arrays of the **aete** resource, along with the equivalent human-language terms. Similarly, an application can use the **aete** resource to describe the parts of each standard suite it supports (if it does not support the entire suite) and any custom OSA events or OSA event object classes defined by the application.

Scripting components can use the information in the **aete** and **aeut** resources in a variety of ways. The next section describes how the Object REXX component uses these resources when it executes or records a script. OSA Event Terminology Resources describes how to create an **aete** resource for your application.

How Object REXX Uses Terminology Information

The Object REXX component uses the information in **aete** resources to construct OSA events from Object REXX statements.

This section describes how the Object REXX component uses the information in the **aeut** and **aete** resources, not how it obtains that information. For a description of the methods available to scripting components for loading information from terminology resources, see Dynamic Loading of Terminology Information.

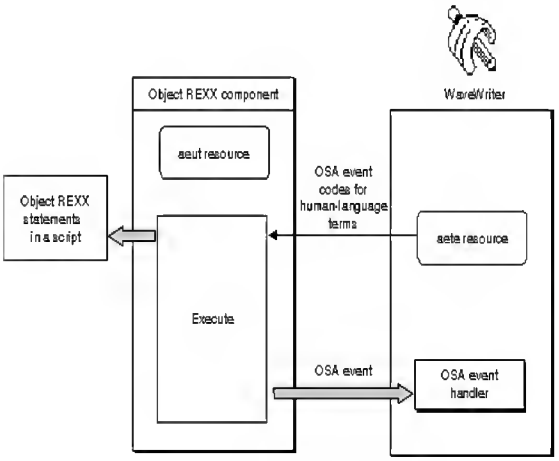
The following figure shows how the Object REXX component uses information from the **aeut** resource and an application's **aete** resource to execute a script consisting of Object REXX statements. When a user executes the script from a script editor, the Object REXX component

translates the script, using information from the **aeut** resource and the application's **aete** resource to map application-specific terms in the script with the equivalent OSA events and OSA event parameters. The Object REXX component evaluates each expression and performs actions or sends OSA events as appropriate.

For example, to execute the following:

```
objspec = document["Sales Report"] chart[1]
app[objspec]~print()
```

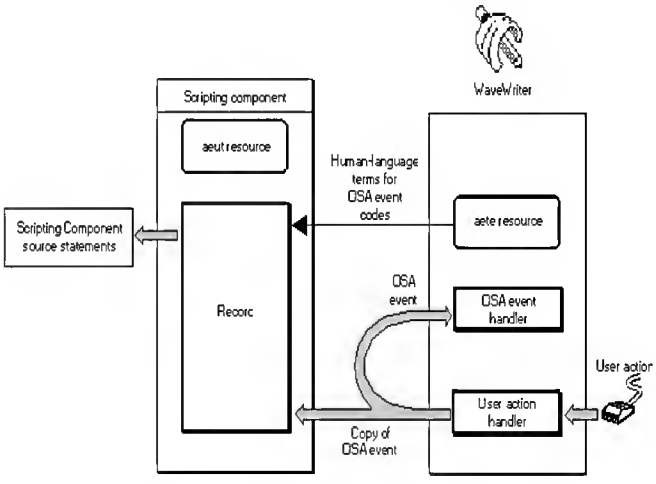
the Object REXX component uses the **aeut** resource and the OS/2 application's **aete** resource to associate the terms used in the script with the Print OSA event, the object class for charts, and the object class for documents. When the Object REXX component executes the statement, it creates and sends a Print event whose direct parameter is an object specifier record that the OS/2 application can resolve as the specified chart. The OS/2 application then handles the OSA event by printing the chart as requested.



Note that although The previous figure shows only one OSA event generated as a result of executing a script, the scripting component could also send a series of OSA events to several different applications, depending on the content of the script.

A recordable application generally needs to be able to send itself a subset of the OSA events that it can handle as a scriptable application. A *recordable event* is an OSA event that any recordable application sends to itself while recording is turned on for the local computer (with the exception of events that the application explicitly identifies as not for recording purposes). For example, after a user turns on recording from the script editor application, the OSA Event Manager sends copies of all recordable events to the script editor. A scripting component previously selected by the user handles each copied event for the script editor by translating the event and recording the translation as part of a script. When a scripting component executes a recorded script, it sends the corresponding OSA events to the applications in which they were recorded.

Every scripting component can choose to support recordable events sent to a recording process (such as a script editor) by receiving them and translating them into useable forms. For example, as shown in the following figure the scripting component uses information from the **aeut** resource and the application's **aete** resource to translate the events into the appropriate human-language terms and display them as scripting statements in the script editor window. When the user opens a recorded script in the script editor and chooses to execute the script, the scripting component sends the OSA events to the OS/2 application, just as in the previous figure.



For example, if the user copies a chart from one document to another and the OS/2 application performs this task by sending itself OSA events, the equivalent source statements in the recorded script would reproduce this action when the script was executed.

To produce these statements, the scripting component uses information from the **aeut** resource and the application's **aete** resource to translate the events sent by the recordable application. The scripting component then sends this equivalent source data to the script editor for display to the user. After completing a recording session, the user can edit and save the resulting script and execute it again at any time.

As shown in the previous two figures, the scripting component uses information it obtains from the **aeut** and **aete** resources when it is creating and recording scripts. Other scripting components might use the same information during compile or decompile, or in other ways that are specific to each component.

Dynamic Loading of Terminology Information

When a scripting component needs information about the user terminology defined in your application's **aete** resource, it sends a Get AETE event to your application. If your application does not handle the Get AETE event, the scripting component reads the terminology information it needs directly from your application's **aete** resource.

Your application does not need to handle the Get AETE event unless it provides separate **aete** resources for plug-in components. If your application does provide separate plug-in components, the Get AETE event allows it to gather terminology information from the **aete** resources for the components that are currently running and add that information to the reply event.

If your application handles the Get AETE event, you must also provide a scripting size resource. A *scripting size resource* is a resource of type **scsz** that provides information about an application's capabilities and preferences for use by scripting components.

To take advantage of dynamic loading, your application must be running. Note that if your application does not provide a handler for the Get AETE event, the scripting component can obtain terminology information directly from your application's **aete** resource even if your application is not running.

Making Your Application Recordable

If you decide to make your application scriptable, you can also make it recordable. A *recordable application* is an application that uses OSA events to report user actions to the OSA Event Manager for recording purposes. A recordable event is any OSA event that a recordable application sends to itself while recording is turned on for the local computer (with the exception of events sent with the kAEDontRecord flag set in the *sendMode* parameter of the AESend function).

When a user turns on recording by clicking the Record button in the Script Editor application, the OSA Event Manager sends copies of all subsequent recordable events to Script Editor. The scripting component handles each copied event for Script Editor by translating it into compiled expressions and recording the compiled expressions as part of a script. (the second figure in section How Object REXX Uses Terminology Information shows how the scripting component uses the **aete** and **aeut** resources when it records a script.) The user can view the equivalent decompiled source data in Script Editor while the script is being recorded. When a user executes a recorded script, the scripting component sends the corresponding OSA events to the applications in which they were recorded.

Applications generally have two parts: the code that implements the application's user interface and the code that actually performs the work of the application when the user manipulates the interface. One way to make your application recordable is to separate these two parts of your application, using OSA events to connect user actions with the work your application performs. This is called *factoring* your application. In a fully factored application, almost all tasks are carried out in response to OSA events. The application translates low-level events that result in significant actions into recordable OSA events and then sends them to itself.

Factoring your application is the recommended method of making your application recordable. However, it is also possible for your application to report user actions by means of OSA events even though it actually performs those actions by some means other than OSA events. You can indicate that you want the OSA Event Manager to record events in this manner, without executing them, by adding the constant kAEDontExecute to the *sendMode* parameter of the AESend function.

Before you decide how to map the user's potential actions to recordable OSA events supported by your application, you need to answer these questions:

- What are the significant (that is, undoable) actions a user can perform with your application that you want to record?
- Which actions can you execute by means of OSA events, and which actions should cause OSA events to be sent but not executed?
- How do you want to record actions that can be described in a scripting language in several different ways?

For example, if your application is a word processor, the user's selection of a range of text should probably not generate an OSA event, because users often select various different pieces of text before deciding to do something to the selection. However, if a user changes the font of a selection, a recordable word processor should generate a corresponding OSA event so that the scripting component can record the change.

In general, a recordable application should generate OSA events for any user action that the user could reverse by choosing **Undo**. A recordable application can usually handle a greater variety of OSA events than it can record, because it must record the same action the same way every time even though OSA events might be able to trigger that action in several different ways.

For more information about recordable applications, factoring, and the OSA Event Manager's recording mechanism, see Recording OSA Events. For a description of the role of the **aete** and **aeut** resources when the Object REXX component records a script, see How Object REXX Uses Terminology Information.

Manipulating and Executing Scripts

Your application can use scripting component routines to manipulate and execute scripts written in any scripting language based on the OSA. This section describes how scripting components use script data and summarizes some of the tasks your application can perform by calling the standard scripting component routines.

Your application can manipulate and execute scripts regardless of whether it is scriptable or recordable. However, if your application is scriptable, you can easily make it capable of manipulating and executing scripts that control its own behavior. For example, the forms application shown in the figure in section Scripting Components and Applications That Execute Scripts uses standard scripting component routines to execute a script whenever the cursor is in the appropriate field and the user presses **Enter** or **Tab**. Applications can also use scripting component routines to allow users to edit, recompile, save, and load such scripts in order to adapt them to their own purposes.

Before using any scripting component routines, your application must open a connection with at least one scripting component. After opening a connection with a component, your application receives a component instance that it can use as the first parameter for any scripting component routine. You can use the Component Manager to establish a connection with the generic scripting component or to establish an explicit connection with any other scripting component. Your application can open connections with different scripting components under different circumstances and, if necessary, simultaneously.

To manipulate or execute scripts written in any scripting language based on the OSA, your application can open a connection with the *generic scripting component* . The generic scripting component in turn attempts to open connections dynamically with the appropriate scripting component for a given script. If your application opens a connection with the generic scripting component, it can load and execute scripts created by any scripting component that is registered with the Component Manager on the current computer. The generic scripting component also provides routines that allow you to determine which scripting component created a particular script and to perform other useful tasks when you are using multiple scripting components.

To manipulate and execute scripts written in a single scripting language only, your application can open an explicit connection with the scripting component for that language. In this case your application can load and execute only those scripts that were created by that component; however, your application can also take advantage of additional routines and other special capabilities provided by the component.

After your application has established a connection with the appropriate scripting component for an existing script, it can use the standard scripting component routines to execute scripts. A script that has not yet been compiled consists of *source data* , or statements in a scripting language. Before executing source data, your application must use scripting component routines to compile it so that the scripting component can keep track of it in memory and execute it.

Scripting components can refer to at least three kinds of *script data* in memory:

- A *compiled script* consists of compiled code that an application can decompile into source data or execute using the standard scripting component routines.
- A *script value* consists of an integer, a string, a Boolean value, constants, PICT data, or any other fixed data returned or used by a scripting component in the course of executing a script.
- A *script context* maintains context information for the execution of other scripts. A script context can also contain executable statements in a scripting language. Like a compiled script, a script context can be decompiled as source data.

  For example, a script context can contain user-defined handlers for specific OSA events. A script context that contains such handlers or other executable statements is called a *script object* . They consist of script statements and have no corresponding entry in OSA event dispatch tables.

Scripting components keep track of script data in memory by means of *script IDs* of type OSAID.

```
typedef unsigned long OSAID;
```

A scripting component assigns a script ID to a compiled script or script context whenever the component creates or loads the corresponding script data. The scripting component routines that compile, load, and execute scripts all return script IDs, and you must pass valid script IDs to many of the other routines that manipulate scripts.

Applications most commonly use scripting component routines to:

- Compile source data and execute the resulting compiled script, so that a user can create a new script and execute it immediately from within the application
- Get a handle to script data in a form that can be saved, and load and execute the script data again when necessary
- Allow users to modify a script, then recompile and save the script
- Redirect OSA events to script contexts

The remainder of this section provides an overview of the scripting component routines you can use to perform these tasks.

Your application can also use scripting component routines to:

- Get information about scripts
- Get information about scripting components
- Coerce script values to descriptor records and vice versa
- Set a resume dispatch function and alternative send, create, and Active functions for use by a scripting component
- Control the recording process directly, turning recording off and on and saving the recorded script for use by your application

Scripting Components provides detailed information about using the scripting component routines.

Compiling, Saving, Modifying, and Executing Scripts

This section introduces some of the scripting component functions your application can use to compile, save, modify, and execute scripts.

To create and execute a script using the Script Editor application, a user can type the script, then press the **Run** button to execute it. Your application can provide similar capabilities by using these functions to compile source data and execute the resulting compiled script:

- The OSACompile function takes a descriptor record with a handle to source data (usually text) and a script ID. If you specify kOSANullScript instead of an existing script ID, OSACompile returns a script ID for the new compiled script, which you can then pass to the OSAExecute function.
- The OSAExecute function takes a script ID for a compiled script and a script ID for a script context, executes the script, and returns a script ID for the resulting script value.

The binding of any global variables in the compiled script is determined by the script context whose script ID you pass to OSAExecute. If you pass kOSANullScript instead of the script ID for a script context, the scripting component provides its own default context. If you want to provide your own script context rather than using the scripting component default context, you can use either OSACompile or OSAMakeContext to create a script context, which you can load and store just like a compiled script.

After creating a script and trying it out, a user may want to save it for future use. Your application should normally save its scripts as script data rather than source data, so that it can reload and execute the data without recompiling it. Before saving script data, you must first call the OSAStore function to get a handle to the data in the form of a descriptor record. You can then save the data to disk.

To allow a user to reload and execute a previously compiled and saved script, your application can call these functions:

- The OSALoad function takes a descriptor record that contains a handle to the saved script data and returns a script ID for the compiled script.
- The OSAExecute function takes a script ID for a compiled script and a script ID for a script context, executes the script, and returns a script ID for the resulting script value.

In most cases you will want to allow users to modify saved scripts and save them again. To allow a user to modify and save a compiled script, your application can call these functions:

- The OSAGetSource function takes a script ID and returns a descriptor record with a handle to the equivalent source data.
- The OSACompile function takes a descriptor record with a handle to source data and a script ID, and returns the same script ID updated so that it refers to the modified and recompiled script.
- The OSAStore function takes a script ID and returns a copy of the corresponding script data in the form of a storage descriptor record.

You can pass the script ID for the compiled script to be modified to the OSAGetSource function, which returns a descriptor record with a handle to the equivalent source data. Your application can then present the source data to the user for editing. When the user has finished editing the source data, you can pass the modified source data and the original script ID to the OSACompile function to update the script ID so that it refers to the modified and recompiled script. Finally, to obtain a handle to the modified script data so you can save it in a resource or

write it to the data fork of a document, you can pass the script ID for the modified compiled script to the OSAStore function.

If your application has no further use for a compiled script or a resulting script value after successfully loading, saving, compiling, or executing a script, you can use the OSADispose function to release the memory assigned to them. The OSADispose function takes a script ID and releases the memory assigned to the corresponding script data. A script ID is no longer valid after the memory associated with it has been released. This means, for example, that a scripting component may assign a different script ID to the same compiled script each time you load it, and that a scripting component may reuse a script ID that is no longer associated with a specific script.

Using Scripting Component Routines provides more information about the standard scripting component routines described in this section.

Using a Script Context to Handle an OSA Event

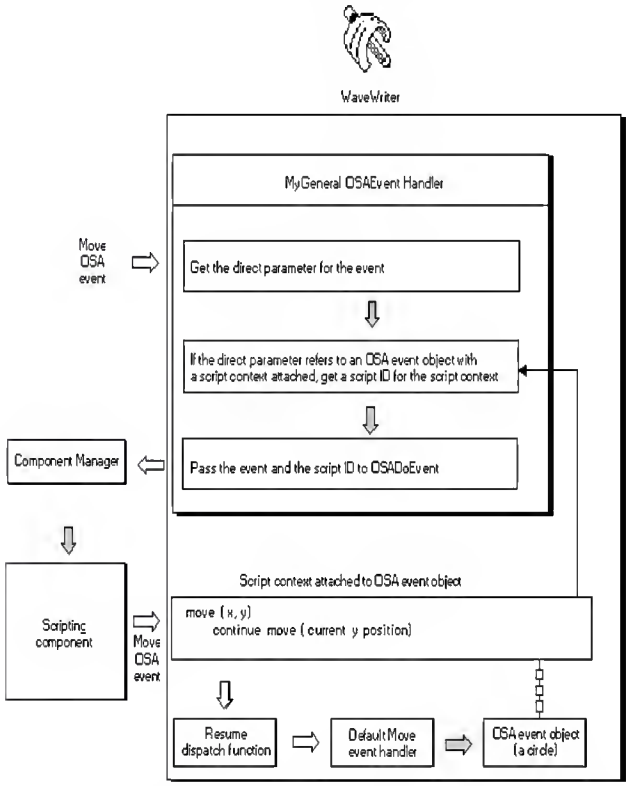
One way to associate a script with an object is to associate a script context with a specific OSA event object-that is, with any object in your application that can be identified by an object specifier record. When an OSA event acts on an OSA event object with which a script context is associated, your application attempts to use the script context to handle the OSA event. This approach can be useful if you want to associate many different scripts with many different kinds of objects.

The following figure illustrates one way that an application can use a script context to handle an OSA event. This example shows how you can use a general OSA event handler to provide initial processing for all OSA events received by your application. If an OSA event acts on an object with which a script context is associated, the general handler attempts to use the script context to handle the event.

The WaveWriter application in the following figure associates script contexts with geometric shapes such as circles or squares. These script contexts can contain one or more user-defined handlers for specific OSA events. For example, the script context shown in the following figure is associated with a circle and contains a handler of this form:

```
move (x, y)
   continue move (current y position)
```

This handler exists only as scripting statements in the script context and does not have an entry in WaveWriter's OSA event dispatch table. WaveWriter does have its own standard OSA event handlers installed in its OSA event dispatch table. When WaveWriter receives a Move event that acts on the circle with which this script context is associated, WaveWriter uses the handler in the script context to modify its own standard handling of the event. The rest of this section describes how this works.



The MyGeneralOSAEventHandler function in the previous figure is installed in WaveWriter's special handler dispatch table. Thus, MyGeneralOSAEventHandler provides initial processing for all OSA events received by WaveWriter. When it receives an OSA event, MyGeneralOSAEventHandler checks whether a script context is associated with the object on which the event acts. If so,

MyGeneralOSAEventHandler passes the event and a script ID for the script context to the OSADoEvent function. If not, MyGeneralOSAEventHandler returns errAEEventNotHandled, which causes the OSA Event Manager to look for the appropriate handler in WaveWriter's OSA event dispatch table.

The OSADoEvent function looks for a handler in the specified script context that can handle the specified event. If the script context does not include an appropriate handler, OSADoEvent returns errAEEventNotHandled. If the script context includes an appropriate handler (in this example, a handler that begins on move), OSADoEvent attempts to use the handler to handle the event.

When it encounters the continue statement during execution of the on move handler shown in the previous figure, the scripting component calls WaveWriter's resume dispatch function. A *resume dispatch function* takes an OSA event and invokes the application's default handler for that event directly, bypassing the application's special handler dispatch table and the MyGeneralOSAEventHandler handler (or its equivalent). In this case, the scripting component uses WaveWriter's default Move handler to move the circle to a different location than the one specified in the original Move event. The location specified by `(x, current y position)` has the same horizontal coordinate as the location specified by the original event, but specifies the circle's original vertical coordinate (the circle's original y position), thus constraining motion to a horizontal direction.

The scripting component calls the resume dispatch function as soon as it encounters a continue statement during script execution. For example, if the handler in the previous figure contained additional indented statements after the continue statement, the scripting component would proceed with the execution of those statements after calling the resume dispatch function successfully.

A script context can modify the event and use the default OSA event handler to execute the modified event, as in this example; or it can override the default handler completely, performing some completely different action; or it can perform some action and then pass the original event to the application's default handler to be handled in the usual way. Script contexts associated with OSA event objects thus provide a way for users to modify or override the way an application responds to a particular OSA event that manipulates those objects.

A general OSA event handler can use the OSAExecuteEvent function instead of OSADoEvent to execute a script context. The main difference between these functions is that OSAExecuteEvent returns the script ID for the resulting script value, whereas OSADoEvent returns a reply event.

To create a script context, pass the source data for the scripting-language statements you want the script context to contain to OSACompile with the *modeFlags* parameter set to kOSACompileIntoContext. The resulting script context is identical to a script context returned by the OSAMakeContext function, except that it contains compiled statements.

Using a Script Context to Handle an OSA Event describes this method of executing a script in more detail.

OSA Event Terminology Resources

This chapter describes the resource structure used by both the **aeut** and **aete** resources and explains how to create an **aete** resource for your application. It also explains how applications that support additional plug-in modules, each with its own **aete** resource, can write a handler for the Get AETE event that collects the **aete** resources from the modules that are currently running.

Before you read this chapter, you should read Introduction to Scripting and the chapters about the OSA Event Manager that are relevant to your application.

The first section in this chapter describes how the Object REXX component interprets Object REXX statements that trigger OSA events. The first section also explains how to define both OSA events and the corresponding user terminology for your application in a way that translates easily into Object REXX statements. If you implement OSA events so that they translate into logical and useful Object REXX scripts, your implementation will probably work well with other scripting components that resemble Object REXX.

The next two sections describe how to do the following:

- Create an **aete** resource
- Handle the Get AETE event

For details about the structure of the data in an **aeut** resource and an **scsz** resource, see Reference to OSA Event Terminology Resources.

Defining Terminology for Use by the Object REXX Component

You should keep two principles in mind when you are defining the OSA event object hierarchy and corresponding terminology for your application:

- Avoid defining new OSA events unless absolutely necessary. For example, instead of defining a custom Find event, use the Get Data event with whose tests. (For more information about whose tests, see Resolving and Creating Object Specifier Records.)

- Use existing object classes or, if you must define your own, define them in a general fashion.

This section describes how the terms you specify in your application's **aete** resource are used in Object REXX instructions that control your application. Before you implement the OSA event object hierarchy for your application, try out your proposed user terminology in Object REXX instructions that use the standard syntax forms described here. This will help you discover some of the advantages and disadvantages of both your proposed object hierarchy and the human-language terminology you are planning to use.

Some Object REXX instructions, such as DO, IF, and SAY keyword instructions, are executed directly by the Object REXX component and do not correspond to OSA events. Others trigger OSA events when the Object REXX component evaluates them. For example, the <<, >>, and == methods of the Object Specifier class correspond to the OSA operators "begins with", "ends with", and "contains", respectively.

In this chapter, the term *script* means a Object REXX program. The Object REXX component interprets the terms used in scripts according to rules the Object REXX language defines. See the *Object REXX Reference for OS/2* for details.

Within a Object REXX script, you can use a scriptable application proxy object to send events to a scriptable application.

Specifically, to trigger OSA events you:

1. Create a scriptable application proxy object (with the CONNECT class method of the ScriptableApp class; ScriptableApp Class provides details)
2. If necessary, code the information for an object reference. (An object reference is usually the first parameter, the direct parameter, of an event message. Object Specifier class provides details about this.)
3. Send event messages (Sending Events describes this.)

Sending Events

In general, the syntax for Object REXX instructions that trigger OSA events follows this pattern:

```
     result=

proxy                        ~event(                          )  ;
        [direct_parameter]             additional_parameters
```

result
The information the event returns. Object REXX converts this string to a list or directory object.

proxy
An instance of the ScriptableApp class. (See ScriptableApp Class for more information.)

direct_parameter
An optional Object Specifier object that further defines the target of the event within the application. (See Object Specifier class for information.)

event
The message to send to the scriptable application.

additional_parameters
An Object REXX directory identifying any additional information needed to process the event. In the application's **aete** resource, a label (string) identifies any *additional_parameter* s. These are optional and can be specified in any order. The directory index should match the label that identifies the parameter in the dictionary. The directory value is the value for the parameter.

ScriptableApp Class

A *scriptable application proxy object* is an instance of the Object REXX ScriptableApp class. Within a Object REXX script, you can use a scriptable application proxy object to send events to a scriptable application.

The CONNECT class method creates a proxy object and connects it to a scriptable application. This connection gives you access to the scriptable application's **aete** resource. You can send messages to the proxy object to control the application, using events, objects, and

properties defined in the dictionary.

Note: Only one scriptable application can be connected at a time.

The ScriptableApp class is a subclass of the Object Class.

CONNECT

```
CONNECT(application_name)
```

This class method returns an instance of the ScriptableApp class that represents the specified scriptable application or returns the NIL object if the connection is unsuccessful.

application_name
The scriptable application.

The scriptable application you have connected remains connected until you disconnect it (with the DISCONNECT class method) or until you connect to another scriptable application. If you connect to a scriptable application while already connected to another scriptable application, the Object REXX component implicitly disconnects the first application before connecting the second application.

The variable initialized before the connection is not changed, even if its name conflicts with a name in the dictionary. That is, only uninitialized variables are used when building references to objects (see Object Specifier class). The DISCONNECT class method, which follows, includes an example.

DISCONNECT

```
DISCONNECT
```

This class method returns `0` if successful in disconnecting the currently connected scriptable application. Otherwise returns `1`.

Variables that contain references to objects are dropped (become uninitialized) if the application is disconnected.

Some examples follow.

Examples

```
font = 'This variable is initialized'

/* The following line creates a proxy for the Lotus 1-2-3** ScriptableApp *
Lotus123 = .ScriptableApp~connect('lotus123')

say font                                    /* Output is 'This variable...'
rc = .ScriptableApp~disconnec                /* Disconnects   */
```

Object Specifier class

In applications, objects can contain other objects (for example, a paragraph contains words). An object contained in another object is called an *element* . A word is an element of a paragraph.

Using an Object REXX concatenation method indicates that the object to the right of the concatenation operator is an element of the object to the left of the operator. You can use the blank (" ") or the || concatenation operator.

The Object Specifier class defines concatenation methods.

An *object reference* identifies an object in an application. It is usually the first parameter, the *direct_parameter* , of an event message.

```
object_class  [   reference_type   ]

  concatenation_operator

   object_class  [   reference_type   ]

concatenation_operator  property_reference  ;
```

reference_type:

```
property
object_class[string]
object_class[ID id_number]
object_class[   position   ]
                FIRST
                MIDDLE
                LAST
                ANY
                ALL
object_class[   BEFORE   object_reference]
                AFTER
object_class[          term   <    term            ]
                 ¬             >
                               <=
                               >=
                               =
                               <<
                               >>
                               ==

                        term   <    term   &
                    ¬          >          |
                               <=
                               >=
                               =
                               <<
                               >>
                               ==
object_class[   number             ,   number            ]
                FIRST                  FIRST
                LAST                   LAST
                MIDDLE                 MIDDLE
                object_reference       object_reference
```

object_class
The name of an object class that the application's dictionary defines. (Examples of object classes are document, paragraph, and table.)

reference_type
The type of object that *object_class* is. The application's dictionary lists the *reference_type* s that the *object_class* supports. The *reference_type* can be one of the following. (Note that object classes do not have to support all of these types.)

Property

An object's property. The *property* is the name of the property.

Example:

```
theFont = word[1] font; /* "font" is the property */
```

Name

The name of an object. The *string* is the name of the object.

Example:

```
my_shoes = table['my shoes']; /* 'my shoes' is the string */
```

Identifier

The unique identifier of an object within the container or application. The *id_number* is an integer that uniquely identifies the object.

Example:

```
window.17 = window[id 17] /* the id_number is 17 */
```

Position

The position of an object within the container. The *position* is the position of the object within the container. A positive integer is the offset from the first element in the container. A negative integer is an offset from the last element.

FIRST	The first element in the container.
MIDDLE	The middle element in the container.
LAST	The last element in the container.
ANY	Any element in the container, such as any odd number or any even number.
ALL	All elements in the container.

Example:

```
any_word = paragraph[2] word[ANY]
/* "2" and "ANY" are positions */
```

Relative Position

The next or previous object relative to a base container.

BEFORE	The target object precedes the *object_reference*.
AFTER	The target object follows the *object_reference*.
object_reference	The reference object that the target object precedes or follows.

Example:

```
the_paragraph = document[1] paragraph[after table[1]];
/* The object_reference is "table" */
```

Test

An object or list of objects whose properties or elements match the specified test criteria or filter.

term	A term used in a comparison.
<	"less than".
>	"greater than".
<=	"less than or equal to".
>=	"greater than or equal to".
=	"equal to".
<<	The OSA operator "begins with".
>>	The OSA operator "ends with".
==	The OSA operator "contains".

&	Logical AND.
|	Logical OR.

Note: A special predefined object *it* is used to iterate over each successive object.

Examples:

```
theWord = word[font='Helvetica' & it << 'A']
/* Tests for word in Helvetica font that begins with A */

firstSea = document[1] word[it = 'Sea' first]
/* Tests for first word that contains 'Sea' */
```

Range

All objects between the two boundary objects.

number	An integer identifying the number of the element in the container.
FIRST	The first element in the container.
MIDDLE	The middle element in the container.
LAST	The last element in the container.
object_reference	A boundary object.

Examples:

```
six_words = word[3, 8]
/* Provides list of 6 words (between word 3 and word 8) */

theText = document[1] text[word]1], paragraph[1]];
/* Provides first word of first paragraph of document 1
/* The boundary objects are not the same type in this example
```

Object properties (variables) are attributes or characteristics of the object. For example, an object of class word has a length property that is set to the number of characters in the word.

property_reference

A property of the object to the left of the *concatenation_operator* . It identifies a property of the containing object reference.

For example:

```
theFont = word[1] font;
```

Structure of OSA Event Terminology Resources

The following table summarizes the resource structure used by both the **aeut** and **aete** resources. Each asterisk (*) in the table indicates the beginning of an array. Each array can contain any number of items, including both additional arrays and specific definitions ( ).

```
    Template version
    Language code
  * Array of suites:
        Suite information
      * Array of events:
            Event information (including information about
              the direct parameter)
          * Array of other parameters:
                Parameter information
      * Array of classes:
            Class description
          * Array of properties:
```

```
            Property information
        * Array of elements:
            Element information
          * Array of key forms:
              Key form information
    * Array of comparison operators:
        Comparison operator information
    * Array of enumerations:
        Enumeration information
      * Array of enumerators:
            Enumerator information
```

The following code fragment shows the resource type declaration in Rez format for the **aeut** resource, which can also serve as a template for an **aete** resource. (Rez is a resource compiler available with the MPW programming environment.) For complete descriptions of all the fields shown in the following code fragment, see Reference to OSA Event Terminology Resources.

```
type 'aeut' {
    hex byte;                                   /*major version in binary-coded
                                                /* decimal (BCD)*/
    hex byte;                                   /*minor version in BCD*/
    integer   Language, english = 0, japanese = 11; /*language code*/
    integer   Script, roman = 0;               /*script code*/
    integer = $$Countof(Suites);
    array Suites {
          pstring;                               /*human-language name of suite
          pstring;                               /*suite description*/
          align word;                           /*alignment*/
          literal longint;                       /*suite ID*/
          integer;                               /*suite level*/
          integer;                               /*suite version*/
          integer = $$Countof(Events);
          array Events {
              pstring;                          /*human-language name of event*/
              pstring;                          /*event description*/
              align word;                        /*alignment*/
              literal longint;                  /*event class*/
              literal longint;                  /*event ID*/
              literal longint noReply = 'null'; /*reply type */
              pstring;                          /*reply description*/
              align word;                        /*alignment*/
              boolean replyRequired,            /*if the reply is*/
                        replyOptional;           /* required*/
              boolean singleItem,              /*if the reply must be a list*/
                        listOfItems;
              boolean notEnumerated,            /*if the type is enumerated*/
                        enumerated;
              boolean reserved;                /*these 13 bits are reserved;*/
              boolean reserved;                /*set them to "reserved"*/
              boolean reserved;
              boolean reserved;
              boolean reserved;
              boolean reserved;
              boolean reserved;
              boolean reserved;
              boolean reserved;
              boolean reserved;
              boolean reserved;
              boolean reserved;
              boolean reserved;
              boolean reserved;
              boolean reserved;
              boolean reserved,                /*if event is verb event or nonve
```

```
          nonVerbEvent;       /*event; used by Japanese diale
literal longint   noParams = 'null'; /*direct param type*/
pstring;                     /*direct param description*/
align word;                   /*alignment*/
boolean directParamRequired, /*if the direct param is require
          directParamOptional;
boolean singleItem,          /*if the param must be a list*/
          listOfItems;
boolean notEnumerated,       /*if the type is enumerated*/
          enumerated;
boolean doesntChangeState,   /*if the event changes server's
          changesState;
boolean reserved;           /*these 12 bits are reserved;*/
boolean reserved;           /*set them to "reserved"*/
boolean reserved;
boolean reserved;
boolean reserved;
boolean reserved;
boolean reserved;
boolean reserved;
boolean reserved;
boolean reserved;
boolean reserved;
boolean reserved;
integer = $$Countof(OtherParams);
array OtherParams {
    pstring;                   /*human-language name for para
    align word;                /*alignment*/
    literal longint;           /*parameter keyword*/
    literal longint;           /*parameter type*/
    pstring;                   /*parameter description*/
    align word;                /*alignment*/
    boolean required,          /*if param is required*/
            optional;
    boolean singleItem,        /*if the param must be a list*/
            listOfItems;
    boolean notEnumerated,     /*if the type is enumerated */
            enumerated;
    boolean isNamed,           /*indicates if this should be th
            isUnnamed;         /* unnamed parameter; only one *
                               /* parameter can be so marked;
                               /* reserved if not required*/
    boolean   reserved;        /*these 9 bits are reserved ; *
    boolean   reserved;        /*set them to "reserved"*/
    boolean   reserved;
    boolean   reserved;
    boolean   reserved;
    boolean   reserved;
    boolean   reserved;
    boolean   reserved;
    boolean   reserved;
    boolean   notFeminine,     /*feminine; set to reserved if
              feminine;        /*required*/
    boolean   notMasculine,    /*masculine; set to reserved if
              masculine;       /*required*/
```

```
        boolean   singular,
                  plural;         /*plural*/
    }
}
integer = $$Countof(Classes);
array Classes {
  pstring;                      /*human-language name for class*/
  align word;                    /*alignment*/
  literal longint;              /*class ID*/
  pstring;                      /*class description*/
  align word;                    /*alignment*/
  integer = $$Countof(Properties);
  array Properties {
      pstring;                    /*human-language name for proper
      align word;                /*alignment*/
      literal longint;            /*property ID*/
      literal longint;            /*property class*/
      pstring;                    /*property description*/
      align word;                /*alignment*/
      boolean reserved;          /*reserved*/
      boolean singleItem,        /*if the property must be a list
              listOfItems;
      boolean notEnumerated,    /*if the type is enumerated*/
              enumerated;
      boolean readOnly,          /*can only read it*/
              readWrite;        /*can read or write it*/
      boolean reserved;          /*these 9 bits are reserved;*/
      boolean reserved;          /*set them to "reserved"*/
      boolean reserved;
      boolean reserved;
      boolean reserved;
      boolean reserved;
      boolean reserved;
      boolean reserved;
      boolean reserved;
      boolean notFeminine,      /*feminine; set to reserved if not
              feminine;          /*required*/
      boolean notMasculine,      /*masculine; set to reserved if n
              masculine;        /*required*/
      boolean singular,
              plural;            /*plural*/
  }
  integer = $$Countof(Elements);
  array Elements {
    literal longint;            /*element class*/
          integer = $$Countof(KeyForms);
          array KeyForms {       /*list of key forms*/
            literal longint
            formAbsolutePosition = 'indx',
            formName = 'name'; /*key form ID*/
          }
    }
}
integer = $$Countof(ComparisonOps);
array ComparisonOps {
```

```
                pstring;                    /*human-language name for*/
                                             /* comparison operator*/
                align word;                  /*alignment*/
                literal longint;            /*comparison operator ID*/
                pstring;                    /*comparison operator description*
                align word;                  /*alignment*/
            }
            integer = $$Countof(Enumerations);
            array Enumerations {             /*list of enumerations*/
                literal longint;            /*enumeration ID*/
                integer = $$Countof(Enumerators);
                array Enumerators {          /*list of enumerators*/
                    pstring;                 /*human-language name for enumer
                    align word;             /*alignment*/
                    literal longint;         /*enumerator ID*/
                    pstring;                 /*enumerator description*/
                    align word;             /*alignment*/
                }
            }
        }
}
```

Creating an OSA Event Terminology Extension Resource

Scriptable applications must include an OSA event terminology extension (**aete**) resource. You use an **aete** resource to inform scripting components about the extent of your application's support for the standard OSA event suites, any custom OSA events or OSA event objects defined by your application, and the corresponding human-language terms for use in scripts that control your application.

The format of the **aete** resource is identical to that of the **aeut** resource, although it serves a different purpose. The **aeut** resource maps human-language names to IDs, keywords, and other codes used in the OSA events described by the current edition of the *OSA Event Registry: Standard Suites* . The **aete** resource for an application uses the same format to accomplish the following:

- Indicate when a set of definitions for a particular suite included in the **aeut** resource is supported in its entirety by the application. For example, an application can indicate that it supports all of the **aeut** resource definitions for the Required and Core suites simply by identifying the suite as a whole in its **aete** resource; the detailed information for each standard suite does not need to be repeated.
- Describe extensions, if any, to the definitions included in the **aeut** resource, such as additional parameters for standard OSA events, additional properties and element classes for standard object classes, and additional key forms for each element class. For example, an application can indicate that it supports all of the definitions for the Required and Core suites included in the **aeut** resource, an additional parameter for one of the core events defined in the **aeut** resource, and an additional property for one of the core object classes defined in the **aeut** resource.
- Describe the standard OSA events and object classes that belong to suites the application does not support in their entirety.
- Describe the application's custom suite-that is, the application's custom OSA events and object classes, if any.

By specifying a suite ID, suite level, and suite version, your application can indicate that it supports an entire suite. Because the **aeut** resource provided by each scripting component lists the human-language terms for all the standard suites, you do not have to repeat this information if you support a suite in its entirety. If you support a subset of a standard suite, you must list all the OSA events, OSA event parameters, object classes, and so on and equivalent human-language terms for the parts of the suite your application does support.

You can include at most one **aete** resource per application or per module. The language code for this resource must match the language code of the language for which you are developing your application. Applications that support additional modules with their own **aete** resources must provide an **scsz** resource and handle the Get AETE event as described in Handling the Get AETE Event.

Important: Each human-language term supported by an application should correspond to a unique OSA event ID, keyword, or other code in either the application's **aete** resource or the **aeut** resource. For example, since the **aeut** resource defines "size" as the human-language equivalent for the property identified by the four-character code **ptsz** (the pPointSize property of text objects), an application's **aete** resource must not define "size" as the human-language equivalent for some other part of an OSA event or object class. However, more than one human-language term can correspond to the same OSA event ID or code. For example, an application's **aete** resource can define a second

human-language term, "point size," that corresponds to the OSA event identifier **ptsz**.

The previous section, Structure of OSA Event Terminology Resources, describes the basic format used by both the **aeut** and **aete** resources.

The sections that follow provide examples of **aete** resources.

Supporting Standard Suites without Extensions

To indicate that your application supports a standard suite in its entirety, without any extensions, your **aete** resource needs to provide only the information that identifies the suite. For example, the following code fragment shows the Rez input for an **aete** resource provided by an application that supports the entire Required and Core suites with no omissions or extensions.

Every **aete** resource must provide the major and minor version numbers for the content of the resource (1 and 0 in the following code fragment) and the language code (English in the following code fragment). For each suite that an application supports in its entirety, without extensions, the **aete** resource provides only the name, suite description, suite ID, suite level (1 for all current suites), and suite version (1 for all current suites). If the **aete** resource provides an empty string as the human-language name for such a suite, a scripting component uses the name provided for the corresponding suite by the **aeut** resource. If an application does not extend or omit any of the definitions in a standard suite, a scripting component can get the rest of the information about the suite-its event and object class definitions, comparison operators, and enumerated groups-from the **aeut** resource. The corresponding arrays in the **aete** resource can therefore be left empty.

Note that the Rez input for resources does not include the align word fields shown in the **aeut** resource type declaration in the previous code fragment. Rez takes care of word alignment automatically.

```
resource aete (0, "JustTwoSuites") {
    1, /*major version in BCD*/
    0, /*minor version in BCD*/
    english, /*language code*/
    roman, /*script code*/
    {   /*array Suites: 2 elements*/
          /*[1]*/
          "", /*human-language name for suite; */
              /* aeut supplies "Required Suite"*/
         "Events that every application should support", /*suite description
          kAERequiredSuite, /*suite code*/
          1, /*suite level*/
          1, /*suite version*/
          {   /*array Events: 0 elements*/
          },
          {   /*array Classes: 0 elements*/
          },
          {   /*array ComparisonOps: 0 elements*/
          },
          {   /*array Enumerations: 0 elements*/
          },
          /*[2]*/
          "", /*human-language name for suite; */
              /* aeut supplies "Core Suite"*/
          "Suite that applies to all applications", /*suite description*/
          kAECoreSuite, /*suite code*/
          1, /*suite level*/
          1, /*suite version*/
          {   /*array Events: 0 element*/
          },
          {   /*array Classes: 0 elements*/
          },
          {   /*array ComparisonOps: 0 elements*/
          },
          {   /*array Enumerations: 0 elements*/
```

```
            }
        }
}
```

Extending the Standard Suites

If, like the **aete** resource shown in the figure in section Supporting Standard Suites without Extensions, your application's **aete** resource indicates that you support an entire standard suite, the scripting component automatically makes use of all the terminology for that suite provided by its **aeut** resource. For this reason, you can easily extend the definitions for a suite that your application supports in its entirety: just provide arrays in the **aete** resource for the definitions not already included in the **aeut** resource. For example, if you are extending the definition of an event to support a single additional parameter, you should provide an array of parameters that includes one item: the description of the new parameter. Similarly, if you are not extending the contents of a particular array, you do not need to include the array in the **aete** resource.

Although an array of descriptions in an **aete** resource need not include descriptions that are already provided by the **aeut** resource, you must include information that defines the position of the array in relation to the other information in the **aete** resource. As the table in section Structure of OSA Event Terminology Resources shows, you can nest the arrays in an **aete** resource within other arrays: for example, an array of parameters is part of the description of an event, and the event description is, in turn, part of the array of event descriptions for a suite.

To add a description of a single new parameter to the definition of an OSA event in a suite that your application supports in its entirety, you need to provide:

- An array of parameters containing one element: the description of the new parameter
- Information that identifies the event definition to which you are adding the parameter
- Information that identifies the suite containing the event

The following code fragment illustrates how this works. This Rez input adds two new parameters ("number of copies" and "print quality") to the Print Documents event in the Required suite, one enumeration (three values for the "print quality" parameter of the Print Documents event) to the Required suite, and a new property ("first indent") to the cParagraph class in the Text suite. It also adds a plural synonym for the cParagraph class: the word "paragraphs."

```
#define keyMyNumberOfCopies 'numc'
#define keyMyPrintQuality   'prtq'
#define typePrintQuality     'pqen'
#define kFast               'fast'
#define kNormal             'nrml'
#define kHighQuality         'hiql'
#define pFirstIndent         'indt'

resource 'aete' (0, "SuiteExtensions") {
    1, /*major version in BCD*/
    0, /*minor version in BCD*/
    english, /*language code*/
    roman, /*script code*/
    {   /*array Suites: 3 elements*/
          /*[1]*/
          "", /*human-language name for suite; */
              /* 'aeut' supplies "Required Suite"*/
          "Events that every application should support", /*suite descripti
          kAERequiredSuite, /*suite code*/
          1, /*suite level*/
          1, /*suite version*/
          {   /*array Events: 1 element*/
              /*[1]*/
              "", /*human-language name for event; */
                  /* 'aeut' supplies "Print Documents"*/
              "Print the specified list of documents", /*event description*
              kCoreEventClass, /*event class*/
```

```
kAEPrintDocuments, /*event ID*/
noReply, /*reply type*/
"", /*reply description*/
replyOptional, /*reply is optional*/
singleItem, /*reply must be single item*/
notEnumerated, /*type is not enumerated*/
reserved, /*these 13 bits are reserved*/
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
typeAlias,           /*direct parameter type*/
"List of documents to print", /*direct parameter description*
directParamRequired, /*direct parameter is required*/
listOfItems,         /*direct parameter must be list*/
notEnumerated,       /*type is not enumerated*/
doesntChangeState,   /*event does not change server's state*/
reserved,             /*these 12 bits are reserved*/
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
{   /*array OtherParams: these fields describe the additional
      /* parameters Number of Copies and Print Quality to the
      /* standard Print Documents event*/
      /*[1]*/
      "number of copies", /*human-language name for parameter
      keyMyNumberOfCopies, /*parameter keyword*/
      typeShortInteger, /*parameter type*/
      "Number of copies to print", /*parameter description*/
      optional, /*parameter is optional*/
      singleItem, /*parameter must be single item*/
      notEnumerated, /*type is not enumerated*/
      reserved, /*these 13 bits are reserved*/
      reserved,
      reserved,
      reserved,
      reserved,
      reserved,
      reserved,
```

```
            reserved,
            reserved,
            reserved,
            reserved,
            reserved,
            reserved,
            /*[2]*/
            "print quality",     /*human-language name for paramete
            keyMyPrintQuality,   /*parameter keyword*/
            typePrintQuality,   /*parameter type*/
            "The quality of the printing",   /*parameter descriptic
            optional,           /*parameter is optional*/
            singleItem,         /*parameter must be single item*/
            enumerated,         /*type is enumerated*/
            reserved,           /*these 13 bits are reserved*/
            reserved,
            reserved,
            reserved,
            reserved,
            reserved,
            reserved,
            reserved,
            reserved,
            reserved,
            reserved,
            reserved,
            reserved
        }
    },
    {   /*array Classes: 0 elements*/
    },
    {   /*array ComparisonOps: 0 elements*/
    },
    {   /*array Enumerations: 1 element*/
        /*these fields add the 'pqen' enumeration to the Required sui
        /*[1]*/
        typePrintQuality, /*enumeration ID*/
        {   /*array Enumerators: 3 elements*/
            /*[1]*/
            "Fast", /*enumerator name*/
            kFast, /*enumerator ID*/
            "Print as quickly as possible", /*enumerator descriptic
            /*[2]*/
            "Normal", /*enumerator name*/
            kNormal, /*enumerator ID*/
            "Print at normal speed", /*enumerator description */
            /*[3]*/
            "High-Quality", /*enumerator name*/
            kHighQuality, /*enumerator ID*/
            "Print at highest quality possible" /*enumerator descri
        }
    },
    /*[2]*/
    "", /*human-language name for suite; */
        /* 'aeut' supplies "Core Suite"*/
```

```
"Suite that applies to all applications", /*suite description*/
kAECoreSuite, /*suite code*/
1, /*suite level*/
1, /*suite version*/
{   /*array Events: 0 elements*/
},
{   /*array Classes: 0 elements*/
},
{   /*array ComparisonOps: 0 elements*/
},
{   /*array Enumerations: 0 elements*/
},
/*[3]*/
"", /*human-language name for suite; */
    /* 'aeut' supplies "Text Suite"*/
"A set of basic classes for text processing", /*suite description
kAETextSuite, /*suite code*/
1, /*suite level*/
1, /*suite version*/
{   /*array Events: 0 elements*/
},
{   /*array Classes: 1 element*/
    /*[1]*/
    "paragraph", /*human-language name for class*/
    cParagraph, /*class ID*/
    "A paragraph", /*class description*/
    {   /*array Properties: 1 element*/
          /*[1]*/
          "first indent",     /*human-language name for property*
          pFirstIndent,       /*property ID*/
          cLongInteger,       /*property class*/
          "First indent of paragraph in points",/*property descri
          reserved,           /*reserved*/
          singleItem,         /*property is single item*/
          notEnumerated,     /*type is not enumerated*/
          readWrite,         /*property can be modified*/
          reserved,           /*these 12 bits are reserved*/
          reserved,
          reserved,
          reserved,
          reserved,
          reserved,
          reserved,
          reserved,
          reserved,
          reserved,
          reserved,
          reserved
    },
    {   /*array Elements: 0 elements*/
    },
    "paragraphs", /*human-language name for class*/
    cParagraph               , /*class ID*/
    "Every paragraph", /*class description*/
    {   /*array Special Properties: 1 element*/
```

```
                    /*[1]*/
                    "", /*human-language name for property*/
                    kAESpecialClassProperties, /*property ID*/
                    cType,                     /*property class*/
                    "",                         /*property description*/
                    reserved,                   /*reserved*/
                    singleItem,                 /*property is single item*/
                    notEnumerated,             /*type is not enumerated*/
                    readOnly,                   /*property cannot be modifi
                    reserved,                   /*these 11 bits are reserve
                    reserved,
                    reserved,
                    reserved,
                    reserved,
                    reserved,
                    reserved,
                    reserved,
                    reserved,
                    reserved,
                    reserved,
                    plural   /*human-language name is plural form*/
                },
                {   /*array Elements: 0 elements*/
                },
            },
            {   /*array ComparisonOps: 0 elements*/
            },
            {   /*array Enumerations: 0 elements*/
            }
    }
}
```

In the previous code fragment, the possible values for the "print quality" parameter belong to an enumeration. This is indicated by the term enumerated in the parameter description. For this reason, the parameter type field contains the ID for the enumeration-typePrintQuality.

The previous code fragment also adds a plural synonym for "paragraph" to the array of classes: the word "paragraphs." Note that this is listed as if it were an additional class, except that it also specifies cParagraph as the class ID. The first property listed for the synonym has property ID kAESpecialClassProperties. This property describes characteristics of the class as a whole; the last flag bit for this property is set to plural, indicating that the term paragraphs is a plural term for the specified class. This property must always be the first property listed for a class. For more information about the kAESpecialClassProperties property, see Property Data.

An enumeration is described only by its ID; its declaration does not include a name or description field. However, a name, value, and description must be provided for each of the enumerators in an enumeration.

You can use the method illustrated in the following figure only to add to the definitions of OSA events and OSA event object classes, not to support subsets of them. For example, to support only a subset of the parameters of an OSA event or only some of the elements or properties of an existing object class, you must list all the definitions from that suite that you do support. The next section provides more information about how to do this.

Human-language names for OSA events, object classes, and so on (including extensions) can include both uppercase and lowercase letters and spaces. For comparison purposes, case does not matter. However, note that the human-language names defined in the following figure are all lowercase. This convention ensures that scripts in which these terms appear will not have capital letters in unexpected places.

Scripting components that get identifiers or strings from user terminology resources are free to change the identifiers or strings as necessary (eliminating spaces, converting identifiers to all uppercase or lowercase, or changing the identifiers altogether) to meet the requirements of a particular task.

Supporting Subsets of Suites

Your application is not required to support all the definitions in a suite. If you wish to support a subset of the definitions in one or more

standard suites, you can collect individual definitions from any number of suites in a placeholder suite whose suite ID is the application's signature or typeWildCard (****). When you support a subset of a suite, you must provide all the definitions you want to support in your **aete** resource.

Supporting New Suites

If your application defines its own custom OSA events or other OSA event constructs, you should include a separate suite section for the suite in the **aete** resource. You should use your application's signature for both the suite ID and the class ID of all events in the suite.

Handling the Get AETE Event

A scripting component sends the Get AETE event to an application when it needs information about the user terminology specified by the application. For example, the Object REXX component sends the Get AETE event when it first attempts to compile a tell statement that specifies a particular application. If your application does not handle the Get AETE event, the scripting component reads the terminology information it needs directly from your application's **aete** resource. Applications that support additional plug-in modules, each with its own **aete** resource, must provide an **scsz** resource and a handler for the Get AETE event that collects the **aete** resources from the modules that are currently running.

If your application does provide separate plug-in modules, the Get AETE event allows it to gather information from the **aete** resources for the modules that are currently running and return the terminology information along with your application's built-in terminology information to the scripting component in the reply event.

Here is a summary of the structure of a Get AETE event:

Get AETE-Get an application's aete resource

Event class	kASObject REXXClass
Event ID	kGetAETE
Required parameter	
Keyword:	keyDirectObject
Descriptor type:	typeInteger
Data:	Language code
Required reply parameter	
Keyword:	keyDirectObject
Descriptor type:	typeAEList or typeAETE
Data:	The application's terminologies
Description	Sent by a scripting component to an application when the scripting component needs information about the application's user terminology

Your application cannot handle the Get AETE event unless it is running. If your application does not provide a handler for the Get AETE event, the scripting component can obtain terminology information directly from your application's **aete** resource even if your application is not running.

If your application handles the Get AETE event, it must also provide a scripting size resource. A scripting size resource is a resource of type **scsz** that provides information about an application's capabilities for use by scripting components. It allows your application to declare whether it needs the Get AETE event and to specify preferences for the sizes of the portion of your application's heap used by a scripting component. For information about the **scsz** resource, see Scripting Size Resource.

A handler for the Get AETE event should perform the following tasks:

- Obtain the language code specified by the event.
- Create a descriptor list to hold the **aete** resources.
- Collect the **aete** resources from all the application's plug-in modules that are currently running, including the application itself, and add them to the list.

- Add the list to the reply OSA event.

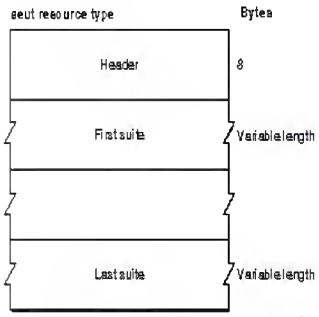
The following code fragment provides an example of a handler for the Get AETE event.

```
OSErr MyGetAETE (OSAEvent theAE, OSAEvent theReply, long refCon)
{
AEDescList     theList;
DescType       returnedType;
Size           actualSize;
INT             languageCode;
OSErr           myErr;

  myErr = errAEEventNotHandled;
  languageCode = 0;

  /* if a reply was not requested, then do not handle */
  if (theReply.dataHandle == NULL)
      return (myErr);

  /* get the language code that Object REXX is requesting so that */
  /* this function can return the aete of a specified language */
  myErr = AEGetParamPtr(&theAE, keyDirectObject, typeLongInteger, &returned
                        &languageCode, sizeof(long), &actualSize);
  if (myErr != noErr)
      return (myErr);

  /* create a list */
  myErr = AECreateList(NULL, 0, FALSE, &theList);
  if (myErr != noErr)
      return (myErr);

  /* get the requested 'aete' resources and put in the list--the */
  /* MyGrabAETE application-defined function does this */
  /* your code should iterate all of your installed code */
  /* extensions and add the aete for each that matches the */
  /* language code requested */
  myErr = MyGrabAETE(languageCode, theList);
  if (myErr != noErr) {
      myErr = AEDisposeDesc(&theList);
      return (myErr);
  }

  /* add list to reply OSA event */
  myErr = AEPutParamDesc(&theReply, keyDirectObject, &theList);
  myErr = AEDisposeDesc(&theList);
  return (myErr);
}
```

The MyGetAETE handler in the previous code fragment begins by setting the function result to errAEEventNotHandled. The function is set to this result if for any reason the handler does not successfully handle the event, so that a system handler provided by the scripting component can at least read the terminology information directly from the application's own **aete** resource. The handler in the previous code fragment then checks the language code specified by the event. After checking to make sure the reply exists, the handler creates a list and uses the application-defined function MyGrabAETE to collect all the appropriate terminology information and append it to the list. The MyGetAETE handler then adds the list to the reply event.

Reference to OSA Event Terminology Resources

The figure in section Structure of OSA Event Terminology Resources shows the complete resource type declaration in Rez format for the **aeut** resource. The same resource structure is used by both the **aeut** and **aete** resources. The following figure shows the format of a compiled **aeut** or **aete** resource.



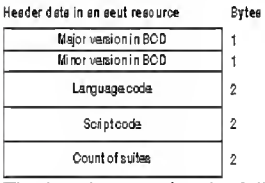
An **aeut** or **aete** resource contains the following:

- Header containing the version and language code of the template and a count of the number of suites the resource describes
- Variable number of suite descriptions

The sections that follow describe the content of the header and each suite description in detail.

Header Data for an OSA Event Terminology Resource

The header for an **aeut** or **aete** resource specifies the version of its contents, the language of the human-language equivalents contained in the resource, a script code, and a count of the number of suites the resource describes. The following figure shows the header format.

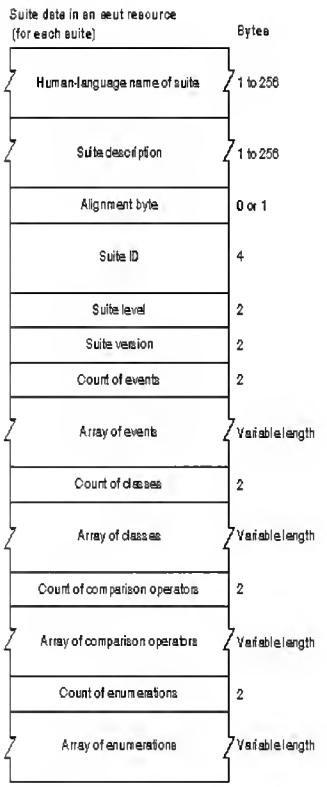


The header contains the following items:

- The major version number of the content of the resource in binary-coded decimal (the major version number for the first release of the **aeut** resource is 1). The major and minor versions describe the content of the resource, not its template. You can use these fields to provide version numbers for the content of your application's **aete** resource.
- The minor version number of the template in binary-coded decimal (the minor version number for the first release of the **aeut** resource is 0).
- The language code for the resource. This code must be the same as the resource ID for the resource.
- The script code for the resource.
- A count of the number of suites described by the resource.

Suite Data for an OSA Event Terminology Resource

Each item in the array of suites for an **aeut** or **aete** resource includes information about the suite ID, level, and version and four arrays that specify the events, object classes, comparison operators, and enumerations for that suite. The following figure shows the format of this suite data.



The data for each suite consists of the following items:

- The human-language name of the suite. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. If the **aete** resource specifies the name as an empty string, the scripting component looks up, in its **aeut** resource, the suite name and other suite data that correspond to the specified suite ID, suite level, and suite version. This strategy simplifies specification of an entire suite and facilitates localization, because the human-language name is provided by the **aeut** resource.

  If the **aete** resource specifies a name other than the name provided by the **aeut** resource for the same suite ID, suite level, and suite version, the scripting component uses the new name with the same suite data from the **aeut** resource. Unless you are defining a custom suite, you should specify an empty string for the name of a suite.

- A human-language description of the suite. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

- A four-character ID that distinguishes the suite from all other suites defined in either the **aeut** or **aete** resources. This value is normally the same as the event class for the OSA events in the suite.

  If the **aete** resource specifies a standard suite name but a suite ID that is different from the suite ID for the standard suite of that name described in the **aeut** resource, the scripting component uses the new suite ID with the standard suite data for the specified name. In general, you should use the standard suite ID for any standard suite that you support.

  If your application uses a custom suite, you should use your application's signature as the event class for the events in the suite and, in addition, as its suite ID. When you register your application's signature with Developer Technical Support, the corresponding event class is automatically registered for your application, and only you can register events that belong to that event class. For information about registering OSA events, contact the OSA Event Registrar.

- The level and version of the suite. For the first version of any suite, the level is usually 1 (indicating that it is the suite that contains the most basic definitions) and the version is 1 (the version of this suite level). More advanced suites (such as a suite for performing more sophisticated text manipulation than the current Text suite allows) will have level numbers greater than 1. All currently defined suites have a level of 1 and a version of 1.

- A count of the events defined for this suite and an array of event definitions.

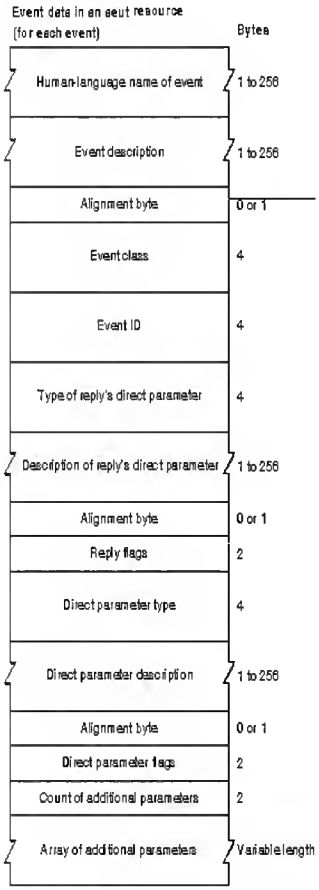
- A count of the object classes defined for this suite and an array of class definitions.

- A count of the comparison operators defined for this suite and an array of comparison operator definitions.

- A count of the enumerations defined for this suite and an array of enumeration definitions.

Event Data

Each item in the array of events for a suite specified in an **aeut** or **aete** resource includes information about the event, the reply, and the direct parameter, and an array that specifies the additional parameters for the event. The following figure shows the format of this event data.



The data for each event consists of the following items:

- The human-language name of the event. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. If the **aete** resource specifies the name as an empty string, the scripting component looks up, in its **aeut** resource, the event name and other event data that correspond to the specified event class and event ID. This strategy facilitates localization, since the human-language name is provided by the **aeut** resource. In this case, the scripting component will use the standard data from the **aeut** resource for the event plus the data provided by the **aete** resource for any additional parameters.

  If the **aete** resource specifies a name other than the name provided by the **aeut** resource for the same event class and event ID, the scripting component uses the new name with the same suite data from the **aeut** resource. You should specify an empty string for the name of any standard event that your application lists explicitly in its **aete** resource.

- A human-language description of the event. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

- The four-character event class for the event. If the **aete** resource specifies a standard event name and an event class other than the event class for the equivalent standard event, the scripting component uses the new event class with the standard event data for the specified name. You should specify the standard event class for any standard event that your application lists explicitly in its **aete** resource.

- The four-character event ID for the event. If the **aete** resource specifies a standard event name and an event ID other than the event ID for the equivalent standard event, the scripting component uses the new event ID with the standard event data for the specified name. You should specify the standard event ID for any standard event that your application lists explicitly in its **aete** resource.

- A four-character descriptor type for the direct parameter of the reply. If the event never needs a reply, or if the reply does not include a direct parameter, this value must be typeNull. Otherwise, the meaning of this field varies according to the values of two of the flags that follow. One flag specifies whether the parameter is a list (singleItem or listOfItems), and the other specifies whether the values for the parameter are enumerated (enumerated or notEnumerated):

  - If the parameter is not a list and its values are not enumerated, this value is the descriptor type for the direct parameter.

  - If the parameter is a list and its values are not enumerated, this value is the descriptor type for each of the items in the

list. (If not all the items in the list are of the same descriptor type, the flag specifying whether the value is a list must have the value singleItem, and the value of this field must be typeAEList.)

- If the parameter is not a list and its values are enumerated, this value is the four-character code for the enumeration defined in either the **aete** or **aeut** resource that contains the allowable values for the parameter. (If the values are enumerated but the enumeration is not defined in either the **aete** or **aeut** resource, the flag specifying whether the parameter's values are enumerated must have the value notEnumerated, and the value of this field must be typeEnumerated.)

- If the parameter is a list and its values are enumerated, this value is the four-character code for the enumeration defined in the same resource that contains the allowable values for all of the items in the list. All items in the list must have one of these enumerated values.

- A human-language description of the direct parameter of the reply. This is a Pascal string that can include any characters. Although the reply may include other parameters, only the direct parameter of the reply is described here. When the resource description is compiled, the resource compiler aligns the string on a word boundary.

- Flags that specify the following as Boolean values:

  - Whether the direct parameter of the reply is required (replyRequired) or optional (replyOptional).

  - Whether the direct parameter of the reply is a single item (singleItem) or a list of items (listOfItems). (See the earlier description of the reply event's four-character descriptor type for information about how this value changes the meaning of the reply type.)

  - Whether named constants, called enumerators, are specified as the only valid values for the direct parameter of the reply (enumerated or notEnumerated). (See the earlier description of the four-character descriptor type for the reply event's direct parameter for information about how this value changes the meaning of the direct parameter type.) For information about specifying enumerators, see Enumeration and Enumerator Data.

  - Following 5 bits are reserved for future use. The values of these bits must be set to reserved.

  - Following 7 bits are reserved for future use as dialect-specific flags. The values of these bits must be set to reserved.

  - Whether the event is a nonverb event (nonVerbEvent). This bit is used by dialects that make this distinction. For all other dialects, set the value of this bit to reserved.

- A four-character descriptor type for the direct parameter of the event. If the event never has a direct parameter, this value must be typeNull. Otherwise, the meaning of this field varies according to the values of two of the flags that follow. One flag specifies whether the parameter is a list (singleItem or listOfItems), and the other specifies whether the values for the parameter are enumerated (enumerated or notEnumerated):

  - If the parameter is not a list and its values are not enumerated, this value is the descriptor type for the direct parameter.

  - If the parameter is a list and its values are not enumerated, this value is the descriptor type for each of the items in the list. (If not all the items in the list are of the same descriptor type, the flag specifying whether the value is a list must have the value singleItem, and the value of this field must be typeAEList.)

  - If the parameter is not a list and its values are enumerated, this value is the four-character code for the enumeration defined in either the **aete** or **aeut** resource that contains the allowable values for the parameter. (If the values are enumerated but the enumeration is not defined in either the **aete** or **aeut** resource, the flag specifying whether the parameter's values are enumerated must have the value notEnumerated, and the value of this field must be typeEnumerated.)

  - If the parameter is a list and its values are enumerated, this value is the four-character code for the enumeration defined in the same resource that contains the allowable values for all of the items in the list. The values of the items in the list must all be one of these enumerated values.

- A human-language description of the direct parameter. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

- Flags that specify the following as Boolean values:

  - Whether the direct parameter of the event is required (directParamRequired) or optional (directParamOptional).

  - Whether the direct parameter of the event is a single item (singleItem) or a list of items (listOfItems). (See the earlier description of the direct parameter's four-character descriptor type for information about how this value changes the meaning of the direct parameter type.)

  - Whether named constants, called enumerators, are specified as the only valid values for the direct parameter (enumerated or notEnumerated). (See the earlier description of the direct parameter's four-character descriptor type for information about how this value changes the meaning of the direct parameter type.) For information about specifying enumerators, see Enumeration and Enumerator Data.

  - Whether receiving this event changes (changesState) or does not change (doesntChangeState) the internal state of

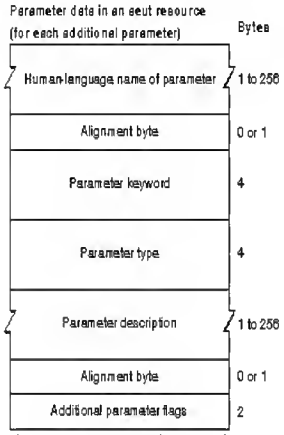
the receiving application. Events that only get information do not change the state of the application, whereas events such as Cut and Move do.

- Following 4 bits are reserved for future use. The values of these bits must be set to reserved.
- Following 8 bits are reserved for future use as dialect-specific flags. The values of these bits must be set to reserved.

- A count of the additional parameters described for this event and an array of additional parameter definitions.

Additional Parameter Data

Each item in the array of additional parameters for an event specified in an **aeut** resource includes information about a single additional parameter. The following figure shows the format of additional parameter data in an **aeut** or **aete** resource.



The data for each additional parameter consists of the following items:

- The human-language name of the parameter. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

  If the **aete** resource specifies the name of an additional parameter as an empty string, the scripting component looks up, in its **aeut** resource, the parameter name and other parameter data that correspond to the specified parameter keyword. If the **aete** resource specifies a name other than the name provided by the **aeut** resource for the same parameter keyword, the scripting component uses the new name with the same parameter data from the **aeut** resource. You should specify an empty string for the name of any standard additional parameter that you list explicitly in an **aete** resource.

- The four-character keyword for the parameter. If the **aete** resource specifies a standard parameter name and a parameter keyword other than the keyword for the equivalent standard parameter, the scripting component uses the new parameter keyword with the standard parameter data for the specified name. You should specify the standard parameter keyword for any standard additional parameter that you list explicitly in an **aete** resource.

- A four-character descriptor type for the parameter. The meaning of this field varies according to the values of two of the flags that follow. One flag specifies whether the parameter is a list (singleItem or listOfItems), and the other specifies whether the values for the parameter are enumerated (enumerated or notEnumerated):

  - If the parameter is not a list and its values are not enumerated, this value is the descriptor type for the direct parameter.
  - If the parameter is a list and its values are not enumerated, this value is the descriptor type for each of the items in the list. (If not all the items in the list are of the same descriptor type, the flag specifying whether the value is a list must have the value singleItem, and the value of this field must be typeAEList.)
  - If the parameter is not a list and its values are enumerated, this value is the four-character code for the enumeration defined in either the **aete** or **aeut** resource that contains the allowable values for the parameter. (If the values are enumerated but the enumeration is not defined in either the **aete** or **aeut** resource, the flag specifying whether the parameter's values are enumerated must have the value notEnumerated, and the value of this field must be typeEnumerated.)
  - If the parameter is a list and its values are enumerated, this value is the four-character code for the enumeration defined in the same resource that contains the allowable values for all of the items in the list. The values of the items in the list must all be one of these enumerated values.

- A human-language description of the parameter. This is a Pascal string that can include any characters. When the resource

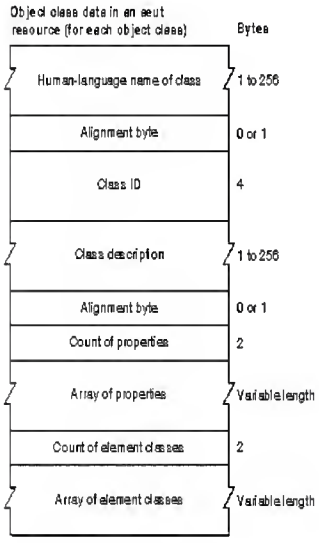
description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

- Flags that specify the following as Boolean values:
    - Whether the parameter is required (required) or optional (optional).
    - Whether the parameter is a single item (singleItem) or a list of items (listOfItems). (See the earlier description of the additional parameter's four-character descriptor type for information about how this value changes the meaning of the parameter type.)
    - Whether named constants, called enumerators, are specified as the only valid values for the parameter (enumerated or notEnumerated). (See the earlier description of the parameter's four-character descriptor type for information about how this value changes the meaning of the parameter type.) For information about specifying enumerators, see Enumeration and Enumerator Data.
    - Whether the parameter is the event's only unnamed parameter (isUnNamed) or is named (isNamed). This bit is used by dialects such as Object REXX Japanese that make this distinction. For all other dialects, set the value of this bit to reserved.
    - Following 4 bits are reserved for future use. The values of these bits must be set to reserved.
    - Following 8 bits are reserved for future use as dialect-specific flags. The values of these bits must be set to reserved.

Extending the Standard Suites includes sample Rez input for an **aete** resource that adds new parameters to a standard OSA event.

Object Class Data

Each item in the array of object classes for a suite includes information about the class and arrays that specify the properties and elements for that class. The following figure shows the format of the object class data in an **aeut** or **aete** resource.



The data for each object class consists of the following items:

- The human-language name of the object class. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

  If the **aete** resource specifies the name of an object class as an empty string, the scripting component looks up, in its **aeut** resource, the class name and other object class data that correspond to the specified class ID. If the **aete** resource specifies a name other than the name provided by the **aeut** resource for the same class ID, the scripting component uses the new name with the same object class data from the **aeut** resource. You should specify an empty string for the name of any standard object class that you list explicitly in an **aete** resource.

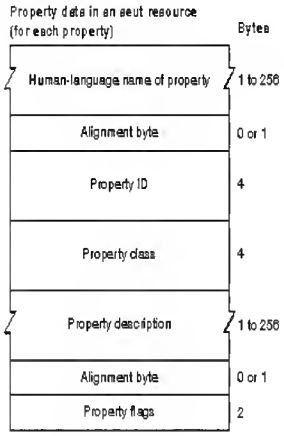
- The four-character class ID for the object class. If the **aete** resource specifies a standard object class name and a class ID other than the class ID for the equivalent standard object class, the scripting component uses the new class ID with the standard object class data for the specified name. You should specify the standard class ID for any standard object class that you list explicitly in an **aete** resource.

- A human-language description of the class. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.
- A count of the properties described for this class and an array of property definitions.
- A count of the element classes described for this class and an array of element class definitions.

To define characteristics of an object class (for instance, whether an object of that class is a single item or a list of items, whether it is singular or plural, and so on), your application's **aete** resource must define a special property of property ID kAESpecialClassProperties as the first property in the array of properties. Because object class data does not include flag bits, the flag bits of this property are used to specify attributes for the class to which the property belongs. The next section describes how this property is defined and used.

Property Data

Each item in the array of properties for an object class includes information about a single property. The following figure shows the format of the property data in an **aeut** or **aete** resource.



The data for each property consists of the following items:

- The human-language name of the property. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

  If the **aete** resource specifies the name of a property as an empty string, the scripting component looks up, in its **aeut** resource, the property name and other property data that correspond to the specified property ID. If the **aete** resource specifies a name other than the name provided by the **aeut** resource for the same property ID, the scripting component uses the new name with the same property data from the **aeut** resource. You should specify an empty string for the name of any standard property that you list explicitly in an **aete** resource.

- The four-character property ID for the property. If the **aete** resource specifies a standard property name and a property ID other than the property ID for the equivalent standard property, the scripting component uses the new property ID with the standard property data for the specified name. You should specify the standard property ID for any standard property that you list explicitly in an **aete** resource.

- A four-character class ID for the object class to which the property belongs. The meaning of this field varies according to the values of two of the flags that follow. One flag specifies whether the property is a list (singleItem or listOfItems), and the other specifies whether the values for the parameter are enumerated (enumerated or notEnumerated):

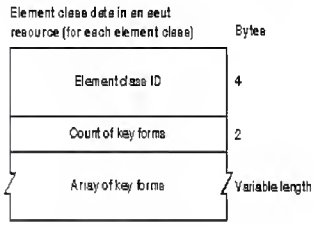
  - If the property is not a list and its values are not enumerated, this value is the class ID for the property.
  - If the property is a list and its values are not enumerated, this value is the class ID for each of the items in the list. (If not all the items in the list are of the same descriptor type, the flag specifying whether the value is a list must have the value singleItem, and the value of this field must be cAEList.)
  - If the property is not a list and its values are enumerated, this value is the four-character code for the enumeration defined in either the **aete** or **aeut** resource that contains the allowable values for the property. (If the values are enumerated but the enumeration is not defined in either the **aete** or **aeut** resource, the flag specifying whether the property's values are enumerated must have the value notEnumerated, and the value of this field must be typeEnumerated.)
  - If the parameter is a list and its values are enumerated, this value is the four-character code for the enumeration defined in the same resource that contains the allowable values for all of the items in the list. The values of the items in the list must all be one of these enumerated values.

- A human-language description of the property. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.
- Flags that specify the following as Boolean values:
    - The first bit is reserved for future use. Its value must be set to reserved.
    - Whether the property is a single item (singleItem) or a list of items (listOfItems). (See the earlier description of the property's four-character class ID for information about how this value changes the meaning of the class ID.)
    - Whether named constants, called enumerators, are specified as the only valid values for the property (enumerated or notEnumerated). (See the earlier description of the property's four-character class ID for information about how this value changes the meaning of the class ID.) For information about specifying enumerators, see Enumeration and Enumerator Data.
    - Whether the property's value can (readWrite) or cannot (readOnly) be set by the Set Data OSA event.
    - Following 4 bits are reserved for future use.
    - Following 5 bits are reserved for future use as dialect-specific flags.
    - Whether the human-language name of the property is feminine (feminine) or not (notFeminine). This bit is used by dialects such as the Object REXX French dialect that make this distinction. For all other dialects, set the value of this bit to reserved.
    - Whether the human-language name of the property is masculine (masculine) or not (notMasculine). This bit is used by dialects such as Object REXX French that make this distinction. For all other dialects, set the value of this bit to reserved.
    - Whether the human-language name of the property is singular (singular) or plural (plural). This bit is used by dialects such as Object REXX French that make this distinction. If you set this bit to reserved, the scripting component will assign it the value singular.

Extending the Standard Suites includes sample Rez input for an **aete** resource that adds a new property to a standard object class. The array of properties in an **aeut** resource begins with a definition of a special property that describes characteristics of the class as a whole using the flags in the definition of that property. A property used in this way to define characteristics of a class must be defined first in the array of properties for that class and must specify kAESpecialClassProperties (**c@#!**) as the property ID, cType as the property class, and an empty string for the property name and property description. If you do not define such a property for a class in your application's **aete** resource, the scripting component will assign that class the default values specified by the first constant for each flag bit in the Rez declaration for the **aeut** resource. (See the figure in section Structure of OSA Event Terminology Resources for the **aeut** resource type declaration.)

Element Class Data

Each item in the array of elements for an object class includes information about a single element class and an array of key forms for that element class. The following figure shows the format of the object class data in an **aeut** or **aete** resource.



The following statements are included for each element class in the array of element classes for an object class:

- The four-character class ID for the element's object class.
- A count of key forms that apply to elements of this class within objects of the class for which these element classes are defined, followed by an array of key forms. Each item in the array must be a value from a special **kfrm** enumeration. (The **aeut** resource includes enumerators for the standard key forms defined in the *OSA Event Registry: Standard Suites* ; an **aete** resource can contain **kfrm** enumerators for additional key forms that are specific to an application. For information about defining enumerators and enumerations, see Enumeration and Enumerator Data.) The enumerators for a **kfrm** enumeration can include **indx** (for the key form formAbsolutePosition), **name** (for the key form formName), **ID** (for the key form formUniqueID), **prop** (for the key form formPropertyID), **rang** (for the key form formRange), **rele** (for the key form formRelativePosition), and **test** (for the key form formTest).

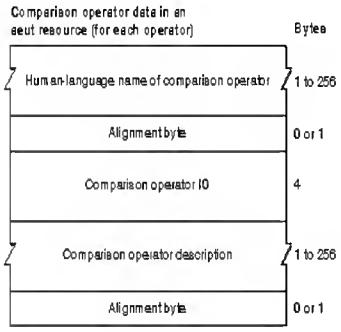
No names or descriptions are provided for element classes, because elements are specified by their object classes, and the declaration of

each object class includes the name and description of the class.

Comparison Operator Data

Each item in the array of comparison operators for a suite includes information about a single comparison operator. The following figure shows the format of the comparison operator data in an **aeut** or **aete** resource.

Note: The Object REXX component currently does not use information about comparison operators. Other scripting components may use this information.



The data for each comparison operator consists of the following items:

- The human-language name of the comparison operator. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

  If the **aete** resource specifies the name of a comparison operator as an empty string, the scripting component looks up, in its **aeut** resource, the comparison operator name and other comparison operator data that correspond to the specified comparison operator ID. If the **aete** resource specifies a name other than the name provided by the **aeut** resource for the same comparison operator ID, the scripting component uses the new name with the same comparison operator data from the **aeut** resource. You should specify an empty string for the name of any standard comparison operator that you list explicitly in an **aete** resource.

- The four-character comparison operator ID for the property. If the **aete** resource specifies a standard comparison operator name and a comparison operator ID other than the comparison operator ID for the equivalent standard comparison operator, the scripting component uses the new comparison operator ID with the standard comparison operator data for the specified name. You should specify the standard comparison operator ID for any standard comparison operator that you list explicitly in an **aete** resource.

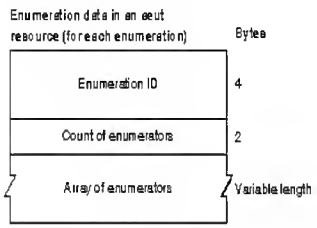
- A human-language description of the comparison operator. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

Extending the Standard Suites includes sample Rez input for an **aete** resource that adds a comparison operator to a standard suite.

Enumeration and Enumerator Data

Each item in the array of enumerations for a suite includes information about a single enumeration and an array of enumerators for that enumeration.

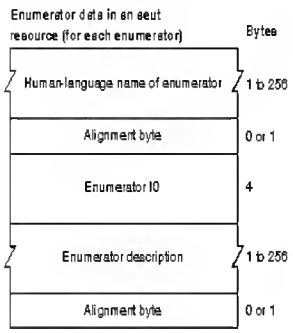
The following figure shows the format of the enumeration data in an **aeut** or **aete** resource.



The data for each enumeration consists of the following items:

- Four-character enumeration ID
- Count of constants, known as enumerators, that specify the allowable values for the enumeration and an array of enumerators

The following figure shows the format of the enumerator data.



The data for each enumerator consists of the following items:

- The human-language name of the enumerator. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary. If the **aete** resource specifies the name of an enumerator as an empty string, the scripting component looks up, in its **aeut** resource, the enumerator name and other enumerator data that correspond to the specified enumerator ID. If the **aete** resource specifies a name other than the name provided by the **aeut** resource for the same enumerator ID, the scripting component uses the new name with the same enumerator data from the **aeut** resource. You should specify an empty string for the name of any standard enumerator that you list explicitly in an **aete** resource.
- The four-character enumerator ID for the enumerator. If the **aete** resource specifies a standard enumerator name and an enumerator ID other than the enumerator ID for the equivalent standard enumerator, the scripting component uses the new enumerator ID with the standard enumerator data for the specified name. You should specify the standard enumerator ID for any standard enumerator that you list explicitly in an **aete** resource.
- A human-language description of the enumerator. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

Extending the Standard Suites includes sample Rez input for an **aete** resource that specifies an enumeration and an array of enumerators.

Scripting Size Resource

If your application handles the Get AETE event, you must provide a scripting size resource. A scripting size resource is an unsigned short value with bit flag definitions that provides information about an application's capabilities for use by scripting components. The following flag settings are available:

kLaunchToGetTerminology	This bit indicates whether the **aete** resource of an application is dynamic and configured by the application when it is launched (for example, the application supports add-on tools which are optionally loaded during initialization). If this bit is set, a scripting component or application must launch the application and issue a Get AETE event to obtain the application's **aete** resource. Otherwise, the application's **aete** resource can be read directly.
kAlwaysSendSubject	When this bit is set, scripting components and other applications that sent events to this application must include a subject attribute in the event.

Recording OSA Events

This chapter describes the general characteristics of a recordable application and provides some examples of how to factor your application for recording. It also provides guidelines to help you decide which user actions to record and how to record them.

Before you read this chapter, you should read the Introduction to Scripting. To factor your application, you must know how to respond to OSA events, create and send OSA events, and resolve and create object specifier records. For comprehensive information about implementing OSA events, see the OSA Events, Responding to OSA Events, Creating and Sending OSA Events, and Resolving and Creating Object

Specifier Records.

The first three sections in this chapter provide the following:

- Description of the basic requirements for recordable applications
- Examples of how to begin factoring your application
- Guidelines for what to record

The fourth section describes how OSA event recording works. You need to read it only if you are developing a script editor, an application that can initiate recording, or a scripting component.

About Recordable Applications

A *recordable application* is an application that uses OSA events to report user actions to the OSA Event Manager for recording purposes. One way to do this is to separate the code that implements your application's user interface from the code that actually performs work when the user manipulates the interface. This is called *factoring* your application. A factored application translates low-level events generated by the user into recordable OSA events that the application sends to itself to perform tasks.

A *recordable event* is any OSA event that any recordable application sends to itself while recording is turned on for the local computer, with the exception of events that are sent with the kAEDontRecord flag set in the *sendMode* parameter of AESend. A *recording process* is any process (for example, the Script Editor application) that can turn recording on and off and can receive and record recordable OSA events.

After OSA event recording has been turned on by a recording process, the OSA Event Manager sends that process copies of all recordable OSA events on the local computer. For example, when a user presses the **Record** button in the Script Editor application, it calls a scripting component routine to turn on recording for the Object REXX component (or any other scripting component). While recording is on, the OSA Event Manager sends Script Editor copies of all subsequent recordable OSA events, which Script Editor records (with the aid of the scripting component) in the form of a compiled script. After turning off recording from Script Editor, the user can edit or execute the recorded script.

Although factoring your application is the recommended method of making your application recordable, it is also possible to report user actions by means of OSA events only when OSA event recording is turned on, even though the application may respond to those actions by some means other than OSA events. In effect, the application uses OSA events to describe user actions without actually using the events to perform the action. To indicate that you want the OSA Event Manager to send a copy of a recordable event to the recording process without actually sending the event to your application, add the constant kAEDontExecute to the *sendMode* parameter of the AESend function.

Even in a factored application, it may not always be possible to send an OSA event that actually executes the task initiated by the user. For example, if the user types some text, it is more practical to use standard routines to draw the text than to send a separate OSA event each time the user presses a key. In this case, the application can draw the text as it is typed in the most convenient manner available; then, when the user finishes typing a sequence of characters-by clicking the mouse button while the cursor is somewhere else in the document or performing some other action-the application can create an OSA event that corresponds to the typing and add the constant kAEDontExecute to the *sendMode* parameter when it sends the OSA event.

If your application needs to know when OSA event recording is turned on and off, it should install handlers for the Recording On and Recording Off events.

Recording On-perform actions associated with beginning of recording session

Event class	kCoreEventClass
Event ID	kAENotifyStartRecording
Parameters	None
Description	Sent by the OSA Event Manager to all running processes on the local computer to inform them that recording has been turned on

Recording Off-perform actions associated with end of recording session

Event class	kCoreEventClass
Event ID	kAENotifyStopRecording
Parameters	None
Description	Sent by the OSA Event Manager to all running processes on the local computer to inform them that recording has been turned off

When a recording process turns on recording, the OSA Event Manager sends all running processes on the local computer a Recording On event. When a user turns off recording, the OSA Event Manager sends all running processes the Recording Off event with the kAEWaitReply flag set. If an application has stored some data (for instance, keystrokes) that needs to be recorded as an OSA event, this is the last chance to send an event for recording purposes. If your application needs to know which recording process has turned recording on or off, it can check the keyOriginalAddressAttr attribute of the Recording On or Recording Off event for the address of the recording process.

Factoring your application is the recommended method of making your application recordable because it guarantees that any action a user can perform via your application's user interface can also be accomplished via OSA events. Factoring also allows you to avoid duplicating code within your application. Instead of using one piece of code to respond to some user action within your application, and another piece of code to respond to the equivalent OSA event, you can use the same code to respond to the OSA event, whether it is sent by your application in response to a user action, by some other application, or by a scripting component in the course of executing a script.

The next section provides some examples of how to go about factoring your application. Regardless of how you factor your application, making it recordable requires you to make decisions about the most useful methods of recording user actions that can be described in several different ways in scripts. If a user moves a window, for example, the window can be described in the corresponding recorded script as window 1, or the window named Fred, or the first window. Although OSA permits recording at a high level and thus avoids many of the problems users encounter with applications that record low-level events such as keystrokes and mouse clicks, scripting components cannot predict what information a user cares about in a given situation. Therefore, a recordable application should send OSA events that correspond to the simplest possible statements in a scripting language. The section What to Record provides some general guidelines for making these kinds of decisions.

How OSA Event Recording Works describes the OSA Event Manager's recording mechanism in more detail, including the role of the Recording On and Recording Off events.

Factoring Your Application for Recording

The recommended way to make your application recordable, or capable of sending OSA events to itself whenever a user performs a significant action, is to factor the code that controls your application's user interface from the code that responds to the user's manipulation of the interface. A fully factored application translates user actions into OSA events that the application sends to itself to initiate tasks.

The examples that follow demonstrate how to factor code that responds to relatively simple user actions such as creating a new document or moving a window. They are intended only to illustrate the general approach you should take; many of the decisions you will need to make while factoring will be unique to your application. What to Record provides guidelines for deciding which user actions to record and how to record them.

If you are factoring an existing application, it is usually a good idea to begin with the required OSA events and any other OSA events that you plan to send in order to execute commands in the **File** menu. You can then proceed to other menu commands and mouse actions. If you are designing a new application and want to make it recordable, you should build factoring into every aspect of your application design.

Factoring the Quit Command and the New Command

This section demonstrates how to factor two **File** menu commands: **Quit** and **New**.

When the user chooses a menu command, an application first determines which one was chosen and then performs the action associated with that command. For example, when a user chooses **Quit** from the **File** menu, an application that is not factored simply calls an application-defined WM_QUIT message. Because Quit Application is one of the required OSA events, it is relatively easy for most applications that support OSA events to factor the code that responds to the **Quit** command.

After a factored application has determined that the user has chosen the **Quit** command, it sends the Quit Application event to itself by calling its MyDoMenuQuit routine.

```
void MyDoMenuQuit ()
{
  OSErr myErr;
  myErr = MySendAEQuit();    /* handle any errors*/
}
```

The MyDoMenuQuit routine, in turn, calls the MySendAEQuit routine, shown in the following code fragment, which creates the Quit Application event and sends it.

```
OSErr MySendAEQuit (DescType saveOpt)
{
  OSAEvent myOSAEvent, defReply;
  OSErr     myErr, ignoreErr;
```

```
  /* create Quit event */
  myErr = AECreateOSAEvent(kCoreEventClass, kAEQuitApplication,
                           &gSelfAddrDesc, kAutoGenerateReturnID,
                           kAnyTransactionID, &myOSAEvent);
    if (myErr == noErr) {
          /* add optional parameter that specifies whether this app */
          /* should prompt user if window is dirty */
          myErr = AEPutParamPtr(&myOSAEvent, keyAESaveOptions, typeEnumerat
                                &saveOpt, sizeof(saveOpt));
    }
    if (myErr = noErr) {
          /*send event */
          myErr = AESend(&myOSAEvent, &defReply, kAENoReply, kAENormalPrior
                         kAEDefaultTimeOut, NULL, NULL);
    }
    ignoreErr = AEDisposeDesc(&myOSAEvent);

    return (myErr);
}
```

After the application receives the Quit Application event, the MyHandleQuit handler shown in the following code fragment performs all the actions associated with that event, such as saving any open documents. (Note that your application should call the ExitToShell procedure from the main event loop, not from your handler for the Quit Application event.)

```
OSErr MyHandleQuit (OSAEvent theOSAEvent, OSAEvent reply, LONG handlerRefcc
{
  BOOL       userCanceled;
  DescType   saveOpt, returnedType;
  Size       actSize;
  OSErr     myErr;

  /* check for missing required parameters */
  myErr = MyGotRequiredParams(theOSAEvent);
  if (myErr == noErr) {
      /* pick up optional save parameter */
      saveOpt = kAEAskUser;     /* the default */
      myErr = AEGetParamPtr(&theOSAEvent, keyAESaveOptions, typeEnumerated,
                            &returnedType, &saveOpt, sizeof(saveOpt), &actS
      if (myErr == errAEDescNotFound)
        myErr = noErr;
      return (myErr);
      if (myErr == noErr) {
          userCanceled = MyPrepareToTerminate(saveOpt);
          if (userCanceled)
            return (kUserCanceled);
      }
  }
  else
      return (myErr);
}
```

If recording has been turned on for a scripting component (for example, after a user clicks the **Record** button in the Script Editor application) and the user quits the application, the OSA Event Manager automatically sends the scripting component a copy of the Quit Application event sent by the MySendAEQuit routine. The scripting component records the event in a compiled script. When a user executes the recorded script, the scripting component sends the same Quit Application event to the application, which calls the MyHandleQuit function and responds to the event just as if the user had chosen **Quit** from the **File** menu.

After you have factored the commands associated with required OSA events for an existing application, you can move on to the other

commands in the **File** menu, such as **New**. After a factored application has determined that the user has chosen **New**, it calls its MyDoMenuNew routine, which sends the Create Element event to the application.

```
void MyDoMenuNew ()
{
  myErr = OSErr;

  myErr = MySendAECreateElement(gNullDesc, cDocument);
  /* handle any errors */
}
```

The container for the new element is the application's default container, specified by a null descriptor record, and the desired class is cDocument. The MyDoMenuNew routine in turn calls the MySendAECreateElement routine shown in the following code fragment, which creates the OSA event and sends it.

```
OSErr MySendAECreateElement (AEDesc cont, DescType elemClass)
{
    OSAEvent myOSAEvent, defReply;
    OSErr     myErr, ignoreErr;

    /* create Create Element event */
    myErr = AECreateOSAEvent(kCoreEventClass, kAECreateElement,
                             &gSelfAddrDesc, kAutoGenerateReturnID,
                             kAnyTransactionID, &myOSAEvent);
    if (myErr == noErr)
          /* add parameter that specifies insertion location for the */
          /* new element */
          myErr = AEPutParamDesc(&myOSAEvent, keyAEInsertHere, &cont);
    if (myErr == noErr)
          /* add parameter that specifies new element's object class */
          myErr = AEPutParamPtr(&myOSAEvent, keyAEObjectClass, typeType,
                                &elemClass, sizeof(elemClass));
    if (myErr == noErr)
          /* send the event */
          myErr = AESend(&myOSAEvent, &defReply, kAENoReply+kAECanInteract,
                         kAENormalPriority, kAEDefaultTimeOut, NULL, NULL);
    ignoreErr = AEDisposeDesc(&myOSAEvent);   /*must dispose of event */
    return (myErr);
}
```

For the purposes of this example, the routine shown in the previous code fragment sends only the required parameters and can only create a new active window with the default name. After the application receives the Create Element event, its MyHandleCreateElement handler performs the requested action, as shown in the following code fragment. In this case, it creates a new active window with a default title.

```
OSErr MyHandleCreateElement (OSAEvent theOSAEvent, OSAEvent reply,
                             long handlerRefCon)
{
    AEDesc     myCont;
    DescType   returnedType, newElemClass;
    Size       actSize;
    DescType   contClass;
    WindowPtr window;
    OSErr     myErr;

    /* get the parameters out of the event */
    /* first get the direct parameter, which specifies insertion */
    /* location for new window--that is, frontmost window */
```

```
    myErr = AEClearDesc(&myCont);
    myErr = AEGetParamDesc(&theOSAEvent, keyAEInsertHere, typeWildCard,
                           &myCont);
    if (myErr == noErr)
          /* get the other required parameter, which specifies class */
          /* cDocument when MyHandleCreateElement creates a new window */
          myErr = AEGetParamPtr(&theOSAEvent, keyAEObjectClass,
                                typeType, &returnedType, &newElemClass,
                                sizeof(newElemClass), &actSize);
    if (myErr == noErr)
          myErr = MyGotRequiredParams(&theOSAEvent);

    if (myErr == noErr) {
          /*check container and class, just to make sure*/
          if ((myCont.descriptorType != typeNull) || (newElemClass != cDocu
              myErr = kWrongContainerOrElement;
          else
              /*MyNewWindow creates a new window with a default name */
              /* and returns a pointer to it in the window parameter*/
              myErr = MyNewWindow(window);
    }
    myErr = AEDisposeDesc(&myCont);
    /*if your app sends a reply in response to the Create Element */
    /* event, then set up the reply event as appropriate*/

    return (myErr);
}
```

If recording has been turned on for a scripting component (for example, after a user clicks the **Record** button in the Script Editor application), the OSA Event Manager automatically sends the scripting component a copy of the Create Element event sent by the MySendAECreateElement routine. The scripting component records the OSA event as a statement in a compiled script. When a user executes the recorded script, the scripting component sends the same Create Element event to the application, which calls its MyHandleCreateElement handler and responds to the event just as if the user had chosen **New** from the **File** menu.

Sending OSA Events without Executing Them

If an application is fully factored, it carries out almost all the tasks a user can perform by sending itself OSA events in the manner illustrated by the listings in the preceding sections. However, in some cases it may not be practical to send an OSA event that actually executes a task performed by the user.

For example, if the user drags a window by its title bar from one position to another, it is inefficient to send a series of OSA events that move the window through a series of positions until the user releases the mouse button. Until the user releases the mouse button, it is not possible to send a single OSA event that drags the window to the new position, because the new position is not yet known. When the WM_WINDOWPOSCHANGED message is received, the window has already been dragged to its new position, and its window record has been updated.

At this point your application can send itself the Set Data event that performs the same action; but to avoid repeating the action that was just performed with DragWindow, you should add the kAEDontExecute constant to the *sendMode* parameter of the AESend function when you send the event. The OSA Event Manager then sends the Set Data event to the recording process, if any, but does not send it to the application.

The following example shows a code fragment from a window procedure that illustrates this approach. The window procedure calls an application-defined routine, MyCreateAESetWindowPos, and the AESend function to create and send a Set Data OSA event that sets the window position to the new location. However, because the window has already been moved, there is no need to execute the Set Data event. To send the event for recording purposes without actually executing it, the MyDoDragWindow routine adds the kAEDontExecute constant to the *sendMode* parameter of the AESend function when it sends the Set Data event.

```
MRESULT EXPENTRY ClientWndProc(HWND hwnd, ULONG msg, MPARAM mp1, MPARAM
```

```
mp2)
{
  switch(msg)
  {
      ...
  case WM_WINDOWPOSCHANGED:
        {
            PSWP pswp = (PSWP)PVOIDFROMMP(mp1); /* window position struct
ptr */

            if(pswp->fl & SWP_MOVE)    /* It is a window drag operation */
            {
              LONG     windowIndex;
              POINTL   newOrigin;
              OSAEvent theOSAEvent, theReply;

              newOrigin.x = pswp->x;
              newOrigin.y = pswp->y;
              MyGetWindowIndex(hwnd, &windowIndex);
              MyCreateAESetWindowPos(windowIndex, &newOrigin, &theOSAEvent)
              AESend(&theOSAEvent,
                      &theReply,
                      kAENoReply + kAECanInteract + kAEDontExecute,
                      kAENormalPriority,
                      kAEDefaultTimeout);
            }
            return WinDefWindowProc(hwnd, msg, mp1, mp2);
        }
        break;
      default:
        return WinDefWindowProc(hwnd, msg, mp1, mp2);
        break;
  }
}
```

If recording has been turned on and the user moves a window, the OSA Event Manager automatically sends the scripting component a copy of the Set Data event sent by the client window procedure but does not send the event to the application. The scripting component records the event as a statement in a compiled script. When a user executes the recorded script, the scripting component sends the same Set Data event to the application. The application's handler for the Set Data event then changes the position of the window.

What to Record

Factoring an application involves making decisions about which user actions generate OSA events, about the content of those events, and about when to send events for recording purposes. For example, the preceding section describes how an application should generate an OSA event that corresponds to a change in the position of a window. Other actions can be more complicated to define in terms of OSA events. This section provides general guidelines for deciding which user actions should generate OSA events and how those events should be defined.

When the user records a series of actions as a script, playing the recorded script back later in exactly the same circumstances must produce exactly the same result. If the circumstances at execution time are similar but not exactly the same as when the script was recorded, the script should also work correctly. However, certain differences will always lead to unexpected results or cause execution to fail.

The goal of these guidelines is to help you create scripts that will work correctly in the largest number of circumstances with the fewest post-recording changes by the user. To accomplish this goal, a recordable application should send itself OSA events that describe as specifically as possible the user's actions in the application's domain without making guesses about the user's intentions.

The way your application uses OSA events to record a user's actions depends in part on the kind of script being recorded. From the user's

perspective, there are at least three kinds of scripts:

- A script application. The icons for these files appear in the Desktop, for example, in the OSA Menu Items folder or the Startup Items folder.
- A script that functions like a menu command, usually acting on the current selection in the current application, and stored either as a compiled script file that appears in the Desktop or as a script stored within an application or one of its documents.
- A script that is "embedded" in an application-that is, explicitly associated with something in a document, such as a field in a form, a cell or row of a spreadsheet, or a button.

The recording guidelines in the sections that follow apply to the recording of scripts that function like menu commands and scripts that are embedded in an application. Because such scripts are executed under a user's direct control, the user expects their execution to cause something to happen, possibly changing the current selection, the Clipboard, or the active window.

The execution of a script application, however, may cause a scripting component to send events to one or more applications intermittently without the user's knowledge. If the script in a script application refers to the current selection, the Clipboard, or the active window, its execution may interfere with other tasks being performed by the user or tasks performed during the execution of other scripts. To create a script application and ensure that it works correctly when executed, a scripter may need to modify the script after it has been recorded.

For example, to eliminate references to the Clipboard, a scripter can use a script variable as a user-defined Clipboard and convert Cut, Copy, and Paste statements to appropriate combinations of Move, Copy, New, and Delete statements, while supplying the previously defined selection as the argument. It may also be necessary to convert a description such as "the front document" to a specific file name or a variable.

Recording User Actions

Two general guidelines apply to the recording of all user actions:

- Send OSA events that correspond to simple statements in a script rather than compound statements.
- Do not record superfluous actions.

In most cases, if the user performs several related actions, your application should send OSA events for each action rather than saving the actions and creating an event that combines them.

For example, if the user selects some text, cuts it, and then pastes it somewhere else, your application should send itself four events that correspond to these actions:

1. Select the text
2. Cut
3. Set the insertion point
4. Paste

Thus, if the user selects characters 5 through 20 of the frontmost document, chooses the **Cut** command from the **Edit** menu, places the insertion point after character 72, and chooses the **Paste** command, your application should send the following events:

- A Select event (event class kAEMiscStandards, event ID kAESelect) with this direct parameter:

```
Keyword                   Descriptor type           Data
keyDirectObject           typeObjectSpecifier       (see indented record)
  keyAEDesiredClass         typeType                  cChar
  keyAEContainer            typeObjectSpecifier       (see indented record)
    keyAEDesiredClass         typeType                  cDocument
    keyAEContainer            typeNull                  No data
    keyAEKeyForm              typeEnumerated            formAbsolutePosition
    keyAEKeyData              typeLongInteger           1
  keyAEKeyForm              typeEnumerated            formRange
  keyAEKeyData              typeRangeDescriptor       (see indented record)
    keyAERangeStart           typeObjectSpecifier       (see indented record)
      keyAEDesiredClass         typeType                  cChar
```

```
    keyAEContainer          typeCurrentContainer    No data
    keyAEKeyForm            typeEnumerated          formAbsolutePosition
    keyAEKeyData            typeLongInteger         5
  keyAERangeStop          typeObjectSpecifier     (see indented record)
    keyAEDesiredClass       typeType                cChar
    keyAEContainer          typeCurrentContainer    No data
    keyAEKeyForm            typeEnumerated          formAbsolutePosition
    keyAEKeyData            typeLongInteger         20
```

- A Cut event (event class kAEMiscStandards, event ID kAECut)
- A Select event with this direct parameter:

```
Keyword                 Descriptor type         Data
keyDirectObject         typeObjectSpecifier     (see indented record)
  keyAEDesiredClass       typeType                cInsertionLoc
  keyAEContainer          typeObjectSpecifier     (see indented record)
    keyAEDesiredClass       typeType                cChar
    keyAEContainer          typeObjectSpecifier     (see indented record)
      keyAEDesiredClass       typeType                cDocument
      keyAEContainer          typeNull                No data
      keyAEKeyForm            typeEnumerated          formAbsolutePosition
      keyAEKeyData            typeLongInteger         1
    keyAEKeyForm            typeEnumerated          formAbsolutePosition
    keyAEKeyData            typeLongInteger         72
  keyAEKeyForm            typeEnumerated          formRelativePosition
  keyAEKeyData            typeEnumerated          kAEAfter
```

- A Paste event

Note: The format used for the direct parameters in this example and throughout this chapter does not show the structure of the direct parameters as they exist within the OSA events. Instead, this format shows what you would obtain after calling AEGetKeyDesc repeatedly to extract the nested descriptor records from the OSA events.

When you call AEGetKeyDesc to extract the descriptor record that specifies an application's default container, AEGetKeyDesc returns a descriptor record of type AEDesc with a descriptor type of typeNull and a data handle whose value is 0.

The first Select event in this example sets the application's pSelection property (that is, the current selection) to the objects identified by the object specifier record in the direct parameter-characters 5 through 20. The second Select event places the insertion point after the object identified by the object specifier in the direct parameter-after character 72.

You could also interpret these four actions as a single Move event that simply moves characters 5 through 20 to after character 72. A user could write such a statement in a script, but for recording purposes four separate events correspond more precisely to the user's actions. For example, if the user performed another paste operation after the first four actions, a Move event would not produce the correct results.

It is equally important for a recordable application not to send superfluous events. For example, your application should not send an event every time the user makes a selection. Instead, it should keep track of the most recent selection made. When the user performs some action on the selection, the application should send an event that sets the selection followed by the event that corresponds to the action taken by the user. However, if the user does not perform an action on the selection, the application should not send an event.

Important: If something is already selected when recording begins, your application should not record that selection. Subsequent user actions should be recorded assuming that there is a selection. By not recording the current selection, you allow the user to record scripts that work, without further modification, much like menu commands that operate on the current selection.

The example just discussed assumes that the application has multiple documents. In such an application, document 1 is always the document in the frontmost window. The examples that follow are simplified, as if they were generated by an application like TeachText that can have only one document open at a time and can therefore locate objects such as characters in the default container. For more complex applications that locate text in cells, documents, and other containers, you must specify additional containers, as appropriate.

Recording the Selection of Text Objects

When your application needs to record a selection that the user has made by dragging through a range of text, it should send itself a Select event that selects a range of characters. For example, a Select event with this direct parameter selects characters 80 through 764:

```
Keyword                  Descriptor type          Data
keyDirectObject          typeObjectSpecifier      (see indented record)
  keyAEDesiredClass        typeType                 cChar
  keyAEContainer           typeNull                 No data
  keyAEKeyForm             typeEnumerated           formRange
  keyAEKeyData             typeRangeDescriptor      (see indented record)
    keyAERangeStart          typeObjectSpecifier      (see indented record)
      keyAEDesiredClass        typeType                 cChar
      keyAEContainer           typeCurrentContainer     No data
      keyAEKeyForm             typeEnumerated           formAbsolutePosition
      keyAEKeyData             typeLongInteger          80
    keyAERangeStop           typeObjectSpecifier      (see indented record)
      keyAEDesiredClass        typeType                 cChar
      keyAEContainer           typeCurrentContainer     No data
      keyAEKeyForm             typeEnumerated           formAbsolutePosition
      keyAEKeyData             typeLongInteger          764
```

It is sufficient to record such a text selection as a range of characters. However, recording selections in other units can make the corresponding scripts easier to read. If you decide to record text selections in other units, keep these guidelines in mind:

- Use the largest whole unit that completely describes the selection.
- Do not mix units.
- Use units appropriate to the method of selection.
- Use logical units rather than units that vary with reformatting.
- Do not try to guess the user's intentions.

The rest of this section provides examples of how to apply these guidelines.

If you do record text selections in units other than characters, record each selection in terms of the largest whole unit that completely describes the selection. For example, suppose the user selects characters 115 through 170 by dragging. Further, suppose the selected characters are exactly the same as words 33 through 50 and also the same as paragraph 2. In this case your application should send itself a Select event with this direct parameter:

```
Keyword                  Descriptor type          Data
keyDirectObject          typeObjectSpecifier      (see indented record)
  keyAEDesiredClass        typeType                 cParagraph
  keyAEContainer           typeNull                 No data
  keyAEKeyForm             typeEnumerated           formAbsolutePosition
  keyAEKeyData             typeLongInteger          2
```

However, if the selected characters do not match a larger unit exactly-for example, if paragraph 2 is larger than the selection or the selection is a portion of two paragraphs-use the largest unit available, in this case words.

For example, a Select event with this direct parameter selects word 33 through word 45:

```
Keyword                  Descriptor type            Data
keyDirectObject          typeObjectSpecifier        (see indented record)
  keyAEDesiredClass        typeType                   cText
  keyAEContainer           typeNull                   No data
  keyAEKeyForm             typeEnumerated             formRange
  keyAEKeyData             typeRangeDescriptor        (see indented record)
    keyAERangeStart          typeObjectSpecifier        (see indented record)
      keyAEDesiredClass        typeType                   cWord
      keyAEContainer           typeCurrentContainer       No data
      keyAEKeyForm             typeEnumerated             formAbsolutePosition
      keyAEKeyData             typeLongInteger            33
      keyAERangeStop           typeObjectSpecifier        (see indented record)
      keyAEDesiredClass        typeType                   cWord
      keyAEContainer           typeCurrentContainer       No data
      keyAEKeyForm             typeEnumerated             formAbsolutePosition
      keyAEKeyData             typeLongInteger            45
```

Do not mix units. You should not send OSA events that define selections like character 2 of word 3 of line 5 of paragraph 2 in document "MyDocument." Instead, define selections as simply as possible; for example, character 45 in the document "MyDocument."

When the user selects text by double-clicking it, your application should send a Select event that specifies words. For example, your application should send a Select event with this direct parameter when the user double-clicks word 5:

```
Keyword                  Descriptor type            Data
keyDirectObject          typeObjectSpecifier        (see indented record)
  keyAEDesiredClass        typeType                   cWord
  keyAEContainer           typeNull                   No data
  keyAEKeyForm             typeEnumerated             formAbsolutePosition
  keyAEKeyData             typeLongInteger            5
```

If the user double-clicks word 5 and then extends the selection through word 9, your application should send a Select event with this direct parameter:

```
Keyword                  Descriptor type            Data
keyDirectObject          typeObjectSpecifier        (see indented record)
  keyAEDesiredClass        typeType                   cText
  keyAEContainer           typeNull                   No data
  keyAEKeyForm             typeEnumerated             formRange
  keyAEKeyData             typeRangeDescriptor        (see indented record)
    keyAERangeStart          typeObjectSpecifier        (see indented record)
```

```
    keyAEDesiredClass        typeType                cWord
    keyAEContainer           typeCurrentContainer    No data
    keyAEKeyForm             typeEnumerated          formAbsolutePosition
    keyAEKeyData             typeLongInteger         5
  keyAERangeStop             typeObjectSpecifier     (see indented record)
    keyAEDesiredClass        typeType                cWord
    keyAEContainer           typeCurrentContainer    No data
    keyAEKeyForm             typeEnumerated          formAbsolutePosition
    keyAEKeyData             typeLongInteger         9
```

If your application supports selection of a paragraph, for example by clicking the left margin, triple-clicking, or some other action, your application should send a Select event that selects the paragraph. For example, a Select event with this direct parameter selects paragraph 2:

```
Keyword                  Descriptor type         Data
keyDirectObject          typeObjectSpecifier     (see indented record)
  keyAEDesiredClass        typeType                cParagraph
  keyAEContainer           typeNull                No data
  keyAEKeyForm             typeEnumerated          formAbsolutePosition
  keyAEKeyData             typeLongInteger         2
```

If your application supports the selection of other units (for instance, cells, rows, and columns in a spreadsheet; embedded graphics in a word processor; or buttons) and if users can select a range of such units, your application should record using those units when appropriate. For example, a Select event with this direct parameter selects row 5 through row 23:

```
Keyword                  Descriptor type         Data
keyDirectObject          typeObjectSpecifier     (see indented record)
  keyAEDesiredClass        typeType                cRow
  keyAEContainer           typeNull                No data
  keyAEKeyForm             typeEnumerated          formRange
  keyAEKeyData             typeRangeDescriptor     (see indented record)
    keyAERangeStart          typeObjectSpecifier     (see indented record)
      keyAEDesiredClass        typeType                cRow
      keyAEContainer           typeCurrentContainer    No data
      keyAEKeyForm             typeEnumerated          formAbsolutePosition
      keyAEKeyData             typeLongInteger         5
    keyAERangeStop           typeObjectSpecifier     (see indented record)
      keyAEDesiredClass        typeType                cRow
      keyAEContainer           typeCurrentContainer    No data
      keyAEKeyForm             typeEnumerated          formAbsolutePosition
      keyAEKeyData             typeLongInteger         23
```

A Select event with this direct parameter selects the second **PICT** image:

```
Keyword                  Descriptor type         Data
```

```
keyDirectObject             typeObjectSpecifier         (see indented record)
  keyAEDesiredClass           typeType                    cPICT
  keyAEContainer              typeNull                    No data
  keyAEKeyForm                typeEnumerated              formAbsolutePosition
  keyAEKeyData                typeLongInteger             2
```

When the user chooses a **Select All** command, your application should send a Select event with this direct parameter to select the contents of the document:

```
Keyword                     Descriptor type             Data
keyDirectObject             typeObjectSpecifier         (see indented record)
  keyAEDesiredClass           typeType                    cProperty
  keyAEContainer              typeNull                    No data
  keyAEKeyForm                typeEnumerated              formPropertyID
  keyAEKeyData                typeLongInteger             pContents
```

Units that vary with reformatting, such as lines and pages in a text document, are not as useful as logical units that describe the data more precisely. Whenever possible, use logical units such as character, word, paragraph, section, and so on.

Do not try to guess the user's intentions. For example, if a selection can be described as either "word 14" or as "the third bold word in paragraph 3," use the simpler description. If you guess the user's intentions, you will be wrong often enough to cause the user to distrust the recording process.

Recording Insertion Points

The insertion point and a selection are synonymous in the Macintosh Operating System. However, scripting languages need a way of specifying a zero-width selection. Sometimes the best way to specify an insertion location is in relation to another object; for example, "after word 5." This section describes recommended methods of specifying an insertion point in a recordable event.

The insertion point can be specified in OSA events by either an insertion location descriptor record (typeInsertionLocation) or an object specifier record (typeObjectSpecifier) that specifies the class cInsertionLoc and the key form formRelativePosition. The Move, Clone, and Create events accept an insertion location descriptor record; other events, including Select and Set Data, require an object specifier record.

Five constants can be used to describe an insertion point in relation to an object or container:

Constant	**Corresponding Insertion Point**
kAEReplace	The specified object will be replaced if not qualified by one of the other phrases
kAEBefore	Just before the specified object (either type typeObjectSpecifier or type typeInsertionLocation)
kAEAfter	Just after the specified object (either type typeObjectSpecifier or type typeInsertionLocation)
kAEBeginning	In the specified container and before all other elements of the same class in that container (type typeInsertionLocation only)
kAEEnd	In the specified container and after all other elements of the same class in that container

For more information about the way Object REXX uses insertion location descriptor records, see Defining Terminology for Use by the Object REXX Component and the *OSA Event Registry: Standard Suite*s . The rest of this section provides examples of object specifier records used to specify insertion points.

Users usually insert objects after some other object. So, unless the insertion point is clearly at the beginning or end of a container or identifies an object to be replaced, use the constant kAEAfter to record the location.

For example, if the user places the insertion point after character 2, your application should send a Select event with this direct parameter:

```
Keyword                  Descriptor type          Data
keyDirectObject          typeObjectSpecifier      (see indented record)
  keyAEDesiredClass        typeType                 cInsertionLoc
  keyAEContainer           typeObjectSpecifier      (see indented record)
    keyAEDesiredClass        typeType                 cChar
    keyAEContainer           typeNull                 No data
    keyAEKeyForm             typeEnumerated           formAbsolutePosition
    keyAEKeyData             typeLongInteger          2
  keyAEKeyForm             typeEnumerated           formRelativePosition
  keyAEKeyData             typeEnumerated           kAEAfter
```

If the selection is not 0 characters wide, the user is replacing the selection with another object, so you can specify the location simply as the object specifier record for the object to be replaced.

If the user clicks the white space after a paragraph somewhere in the middle of the document, defining the insertion point becomes more complex because different applications deal with this situation in different ways. Some place the insertion point at the end of the current paragraph, while others place the insertion point at the beginning of the next paragraph. Depending on the way your application handles this situation, you should use an object specifier record that specifies either kAEBeginning or kAEEnd.

Remember that the Select event requires an object specifier record. Thus, if you want to place the insertion point at the beginning of a paragraph, use an object specifier record that specifies a location just before the first item of the paragraph, rather than an insertion location descriptor record.

For example, a Select event with this direct parameter places the insertion point just before the first item of paragraph 3:

```
Keyword                  Descriptor type          Data
keyDirectObject          typeObjectSpecifier      (see indented record)
  keyAEDesiredClass        typeType                 cInsertionLoc
  keyAEContainer           typeObjectSpecifier      (see indented record)
    keyAEDesiredClass        typeType                 cItem
    keyAEContainer           typeObjectSpecifier      (see indented record)
      keyAEDesiredClass        typeType                 cParagraph
      keyAEContainer           typeNull                 No data
      keyAEKeyForm             typeEnumerated           formAbsolutePosition
      keyAEKeyData             typeLongInteger          3
    keyAEKeyForm             typeType                 formAbsolutePosition
    keyAEKeyData             typeLongInteger          1
  keyAEKeyForm             typeEnumerated           formRelativePosition
  keyAEKeyData             typeEnumerated           kAEPrevious
```

If the user clicks the left edge of the first line in a paragraph, thus setting the insertion point before the beginning of the paragraph, you should use a similar strategy. However, this is the only situation in which you should use kAEPrevious.

When the insertion point is at the end of a document record, use an object specifier record that specifies the location after the last item in the document.

For example, a Select event with this direct parameter places the insertion point just after the last item in a document:

```
Keyword                  Descriptor type          Data
keyDirectObject          typeObjectSpecifier      (see indented record)
```

```
keyAEDesiredClass          typeType                  cInsertionLoc
keyAEContainer             typeObjectSpecifier       (see indented record)
  keyAEDesiredClass          typeType                  cItem
  keyAEContainer             typeNull                  No data
  keyAEKeyForm               typeEnumerated            formAbsolutePosition
  keyAEKeyData               typeLongInteger           -1
keyAEKeyForm               typeEnumerated            formRelativePosition
keyAEKeyData               typeEnumerated            kAENext
```

A Select event with this direct parameter places the insertion point just after the last item in paragraph 3:

```
Keyword                    Descriptor type           Data
keyDirectObject            typeObjectSpecifier       (see indented record)
  keyAEDesiredClass          typeType                  cInsertionLoc
  keyAEContainer             typeObjectSpecifier       (see indented record)
    keyAEDesiredClass          typeType                  cParagraph
    keyAEContainer             typeNull                  No data
    keyAEKeyForm               typeEnumerated            formAbsolutePosition
    keyAEKeyData               typeLongInteger           3
  keyAEKeyForm               typeEnumerated            formRelativePosition
  keyAEKeyData               typeEnumerated            kAENext
```

Recording Typing

In general, to record typing your application should send itself a Set Data event that sets the contents of the selection. The data should be unstyled text. When your application handles the Set Data event, it should apply the styles that prevail at the insertion point. If your application supports styled text, you need to decide how to apply styles to new text and how to record style changes to selected text. Follow these general guidelines for recording typing:

- When the user sets an insertion point and types new text, use the styles defined for the text just before the insertion location.
- When a user selects text and changes its style, apply the changes to the selection.
- If a user types or pastes new text into a selection, place the insertion point after the new text.

The rest of this section provides examples of how to apply these guidelines.

Suppose the user sets an insertion point and then types something. Your application should use the style, font, size, and other characteristics of the text just before the insertion point for the new text, and it should record only the new characters inserted. For example, to place the insertion point after word 30 and insert the text "This is the new text," your application can send a Select event followed by a Set Data event:

- A Select event with this direct parameter places the insertion point after word 30:

```
Keyword                    Descriptor type           Data
keyDirectObject            typeObjectSpecifier       (see indented record)
  keyAEDesiredClass          typeType                  cInsertionLoc
  keyAEContainer             typeObjectSpecifier       (see indented record)
```

```
    keyAEDesiredClass          typeType                  cWord
    keyAEContainer             typeNull                  No data
    keyAEKeyForm               typeEnumerated            formAbsolutePosition
    keyAEKeyData               typeLongInteger           30
  keyAEKeyForm                 typeEnumerated            formRelativePosition
  keyAEKeyData                 typeEnumerated            kAENext
```

- A Set Data event (event class kAECoreSuite, event ID kAESetData) with these parameters (keyDirectObject and keyAEData) sets the selection to the new text:

```
Keyword                    Descriptor type           Data
keyDirectObject            typeObjectSpecifier       (see indented record)
  keyAEDesiredClass          typeType                  cProperty
  keyAEContainer             typeObjectSpecifier       (see indented record)
    keyAEDesiredClass          typeType                  cProperty
    keyAEContainer             typeNull                  No data
    keyAEKeyForm               typeEnumerated            formPropertyID
    keyAEKeyData               typeLongInteger           pSelection
  keyAEKeyForm               typeEnumerated            formPropertyID
  keyAEKeyData               typeType                  pContents
keyAEData                  typeChar                  "This is the new text"
```

Notice that the Select event in this example causes your application to set its pSelection property (the current selection) to the location specified by the object specifier record in the direct parameter-that is, after word 30. The Set Data event then sets the contents of the selection to a text string. The pContents property specified by the object specifier record in the direct parameter of the Set Data event represents the contents of the selection, and the text string in the keyAEData parameter is the text to which the selection's contents is to be set.

At this stage, the insertion point is after word 35-the last word added by typing. If the user now selects one of the new words, say word 34, and changes the style to boldface and the font to Helvetica, send a Select event and two Set Data events to record the action:

- A Select event with this direct parameter selects word 34:

```
Keyword                    Descriptor type           Data
keyDirectObject            typeObjectSpecifier       (see indented record)
  keyAEDesiredClass          typeType                  cWord
  keyAEContainer             typeNull                  No data
  keyAEKeyForm               typeEnumerated            formAbsolutePosition
  keyAEKeyData               typeLongInteger           34
```

- A Set Data event with these parameters sets the style of the selection to boldface:

```
Keyword                    Descriptor type           Data
keyDirectObject            typeObjectSpecifier       (see indented record)
  keyAEDesiredClass          typeType                  cProperty
  keyAEContainer             typeObjectSpecifier       (see indented record)
    keyAEDesiredClass          typeType                  cProperty
```

```
    keyAEContainer          typeNull                No data
    keyAEKeyForm            typeEnumerated          formPropertyID
    keyAEKeyData            typeLongInteger         pSelection
  keyAEKeyForm              typeType                formPropertyID
  keyAEKeyData              typeType                pTextStyles
keyAEData                   typeEnumerated          kAEBold
```

- A Set Data event with these parameters sets the font of the selection to Helvetica:

```
Keyword                     Descriptor type         Data
keyDirectObject             typeObjectSpecifier     (see indented record)
  keyAEDesiredClass           typeType                cProperty
  keyAEContainer              typeObjectSpecifier     (see indented record)
    keyAEDesiredClass           typeType                cProperty
    keyAEContainer              typeNull                No data
    keyAEKeyForm                typeEnumerated          formPropertyID
    keyAEKeyData                typeLongInteger         pSelection
  keyAEKeyForm                typeEnumerated          formPropertyID
  keyAEKeyData                typeType                pFont
keyAEData                   typeChar                "Helvetica"
```

After these three events are sent, word 34 remains selected. Thus, subsequent user actions upon the same selection do not require your application to send an additional event to set the selection. Your application should maintain the selection as long as the selected text is not replaced. If the user types or pastes new text into the selection, your application should place the insertion point after the new text.

Such a strategy might result in a series of events like these:

- A Set Data event with these parameters sets the contents of a selection to "More new text":

```
Keyword                     Descriptor type         Data
keyDirectObject             typeObjectSpecifier     (see indented record)
  keyAEDesiredClass           typeType                cProperty
  keyAEContainer              typeObjectSpecifier     (see indented record)
    keyAEDesiredClass           typeType                cProperty
    keyAEContainer              typeNull                No data
    keyAEKeyForm                typeEnumerated          formPropertyID
    keyAEKeyData                typeLongInteger         pSelection
  keyAEKeyForm                typeEnumerated          formPropertyID
  keyAEKeyData                typeType                pContents
keyAEData                   typeChar                "More new text"
```

- Two Paste events paste the contents of the Clipboard twice after the new text.

Recording the Selection of Nontext Objects

The selection of nontext objects differs from the selection of text objects mainly in the way a recordable application specifies the objects. For example, if the user is working in a table or spreadsheet and selects row 5, column 3, your application can send a Select event with this direct parameter:

```
Keyword                    Descriptor type            Data
keyDirectObject            typeObjectSpecifier        (see indented record)
  keyAEDesiredClass          typeType                   cRow
  keyAEContainer             typeObjectSpecifier        (see indented record)
    keyAEDesiredClass          typeType                   cColumn
    keyAEContainer             typeNull                   No data
    keyAEKeyForm               typeEnumerated             formAbsolutePosition
    keyAEKeyData               typeLongInteger            3
  keyAEKeyForm               typeEnumerated             formAbsolutePosition
  keyAEKeyData               typeLongInteger            5
```

When recording a range of cells, use a range of rows through a range of columns.

For example, if the user selects row 5 column 3 through row 6 column 4, specify columns 3 through 4 of rows 5 through 6 by sending a Select event with this direct parameter:

```
Keyword                    Descriptor type            Data
keyDirectObject            typeObjectSpecifier        (see indented record)
  keyAEDesiredClass          typeType                   cRow
  keyAEContainer             typeObjectSpecifier        (see indented record)
    keyAEDesiredClass          typeType                   cColumn
    keyAEContainer             typeNull                   No data
    keyAEKeyForm               typeEnumerated             formRange
    keyAEKeyData               typeRangeDescriptor        (see indented record)
      keyAERangeStart            typeObjectSpecifier        (see indented record)
        keyAEDesiredClass          typeType                   cColumn
        keyAEContainer             typeCurrentContainer       No data
        keyAEKeyForm               typeEnumerated             formAbsolutePosition
        keyAEKeyData               typeLongInteger            3
    keyAERangeStop             typeObjectSpecifier        (see indented record)
        keyAEDesiredClass          typeType                   cColumn
        keyAEContainer             typeCurrentContainer       No data
        keyAEKeyForm               typeEnumerated             formAbsolutePosition
        keyAEKeyData               typeLongInteger            4
  keyAEKeyForm               typeEnumerated             formRange
  keyAEKeyData               typeRangeDescriptor        (see indented record)
    keyAERangeStart            typeObjectSpecifier        (see indented record)
      keyAEDesiredClass          typeType                   cRow
```

```
    keyAEContainer          typeCurrentContainer      No data
    keyAEKeyForm            typeEnumerated            formAbsolutePosition
    keyAEKeyData            typeLongInteger           5
  keyAERangeStop          typeObjectSpecifier       (see indented record)
    keyAEDesiredClass       typeType                  cRow
    keyAEContainer          typeCurrentContainer      No data
    keyAEKeyForm            typeEnumerated            formAbsolutePosition
    keyAEKeyData            typeLongInteger           6
```

In some drawing and layout applications, users are used to dealing with insertion points at specific locations rather than relative to other objects. For example, setting an insertion point in a recordable drawing application might cause the application to send itself a Select event that places the insertion location at (235, 330)-that is, the location defined by a vertical coordinate of 235 and a horizontal coordinate of 330. A Select event that does this could have this direct parameter:

```
Keyword                 Descriptor type         Data
keyDirectObject         typeObjectSpecifier     (see indented record)
  keyAEDesiredClass       typeType                cInsertionLocation
  keyAEContainer          typeNull                No data
  keyAEKeyForm            typeEnumerated          formAbsolutePosition
  keyAEKeyData            typeAEList              (see indented record)
                            typeLongInteger         235
                            typeLongInteger         330
```

Notice that the key data corresponds to an application's extension of the standard interpretation of key form formAbsolutePosition.

To set a selection that consists of noncontiguous objects, an application should send events that correspond to statements like these:

```
select {¬
    row 5 thru 6 of column 3 thru 4, ¬
    row 22 of column 6}

select {circle 2, rectangle 12, text frame 2}

select {file "Guidelines", file "Test Results"}
```

Identifying Objects

The way a recordable application identifies objects can involve assumptions about the user's criteria for selecting those objects. In general, such assumptions should be avoided. Follow these guidelines for identifying objects:

- If you are not absolutely certain of the user's criteria for selecting an object, identify the object by name.
- If the object does not have a name, identify it by index.
- Determine the index based on the order in which a user would see the objects when reading a document.
- Identify windows and open documents on which actions are taken as the frontmost window or document.

The rest of this section provides examples of how to apply these guidelines.

Suppose a user is working with an electronic mail application that permits a variety of sorting methods for messages received. If the user is currently looking at messages sorted by date and then deletes the second message in the list, that message should be identified by name rather than by date. Use an object's name in any situation where it is not completely clear which identifying criteria the user had in mind.

Suppose a user has used the application's **Find** command to locate all messages created on a certain date. In this case, it might be appropriate to identify "every message whose creation date..." in the corresponding OSA event. However, if the user did not ask for all messages created on that date, you cannot be sure whether the user really wanted every message or only a particular one. For instance, perhaps the user could not remember the name, but only an approximate date. In this case a recordable application should identify the message by name.

Just as names are more specific and usually more desirable than whose tests, names are usually more specific and more readable than identifiers or indices. However, some objects may not have a name, only some other identifier or an index. Even though an identifier is more specific than an index, a logically defined index of position is more readable and is therefore recommended. For example, if a document contains unnamed illustrations, the user is more likely to identify a figure by index (order from the beginning of the file) than identifier (such as order created).

Suppose a document contains two figures that appear at first glance to be side by side, except that the right one is slightly taller and therefore begins higher on the page than the left one. In cases such as this, your application should determine the index based on the order in which the user would see the objects when reading a document. For Roman script systems, this means reading from left to right and from top to bottom. In the example just described, the leftmost, shorter figure would have a lower figure number than the rightmost, taller one.

When your application needs to refer to a window or a document, it should identify the object with an object specifier record that corresponds to the first, or front, window:

```
Keyword                  Descriptor type          Data
keyDirectObject          typeObjectSpecifier      (see indented record)
  keyAEDesiredClass        typeType                 cWindow
  keyAEContainer           typeNull                 No data
  keyAEKeyForm             typeEnumerated           formAbsolutePosition
  keyAEKeyData             typeLongInteger          1
```

This strategy allows users to record scripts that will work on any window, regardless of its name. Similarly, events that act on an open document should identify it as "document 1."

It is usually possible to describe objects in several different ways. If an object has a unique name, use that. For example, instead of an object specifier record that describes "column number 7," use one that describes "the column named 'March'":

```
Keyword                  Descriptor type          Data
keyDirectObject          typeObjectSpecifier      (see indented record)
  keyAEDesiredClass        typeType                 cColumn
  keyAEContainer           typeNull                 No data
  keyAEKeyForm             typeEnumerated           formName
  keyAEKeyData             typeLongInteger          "March"
```

It may be that such an object could also be described in a more complex manner, such as picture 1 of paragraph 302 of chapter 2. But complex descriptions like this should be used only as a last resort if no simpler name is available.

In general, be as specific as possible when you identify a selection in a recordable event. The user can generalize, as necessary, by editing the recorded script.

Moving the Selection during Recording

If recording is turned on and the user makes a selection, performs some action, and then makes a different selection, your application must make a decision: should it record the second selection in absolute terms or relative to the first selection? That is, should the corresponding script statement be:

```
select insertion location before paragraph 5
```

or

```
select insertion location before paragraph after selection
```

Both statements may be appropriate under different conditions. But suppose that the user had selected paragraph 3 and now selects paragraph 12 or picture 3. Relative addressing does not make sense in these situations because the distance involved is too great or the unit is different. When you cannot be sure of the user's intent, you should use absolute addressing. You can safely use relative addressing only when the user moves the selection or insertion point by only one unit, as with the arrow keys.

Even the use of the arrow keys does not guarantee that you can use relative addressing. For example, suppose that the user has selected cell 5 of row 2 in a spreadsheet and then presses the **Left Arrow** key three times. In this case, it is best to send OSA events equivalent to the statement

```
select cell 3 of row 2
```

rather than the statements

```
select the cell before selection
select the cell before selection
select the cell before selection
```

Using relative addressing in certain circumstances may minimize the amount of editing that the user must do after recording a script. However, recordable applications are not required to use relative addressing.

Recording Interactions with Dialog Boxes

When executing scripts, users normally do not want to see dialog boxes. Therefore, your application should record information specified by the user in dialog boxes rather than sending events that would cause the dialog boxes to appear during script execution.

For example, suppose a user chooses the **Close** command and the standard save changes dialog box appears. If the user then clicks **Save**, your application should send a Close event that corresponds to a statement like this:

```
close document "MyDoc" saving Yes
```

Any settings in a dialog box that the user does not change (such as the range of pages to print in a Print dialog box) should not be recorded.

How OSA Event Recording Works

Scripting components use the OSA Event Manager's recording mechanism to allow a recording process such as the Script Editor application to control recording into scripts. Script editors and applications that provide their own recording capabilities can take advantage of the recording mechanism via standard scripting component routines.

This section describes how scripting components use OSA event recording. You need to read this section if you are developing a scripting component or a script-editing application, or if you want your application to initiate and control OSA event recording. For information about using the standard scripting component routines to turn recording off and on, see Recording Scripts.

When a user turns on recording for a recording process (for example, by clicking the **Record** button in Script Editor), the recording process calls a scripting component routine (OSAStartRecording) to turn recording on. The scripting component responds by sending a Start Recording event to the recording process (or any running process on the local computer that has been registered with the OSA Event

Manager using the AEInit function).

Start Recording-begin sending copies of recordable events to recording process.

Event class	kCoreEventClass
Event ID	kAEStartRecording
Parameters	None
Description	Sent by a scripting component to the recording process (or to any running process on the local computer that has been registered with the OSA Event Manager using the AEInit function), but handled by the OSA Event Manager. The OSA Event Manager responds by turning on recording and sending a Recording On event to all running processes on the local computer that have been registered with the OSA Event Manager using the AEInit function. This event must be addressed using a process ID (PID); it should never be sent to an address specified as kCurrentProcess.

The recording process should not handle the Start Recording event. Instead, the OSA Event Manager handles it by sending a Recording On event to all running processes on the local computer that have been registered with the OSA Event Manager using the AEInit function and sending copies of all subsequent recordable events to the recording process. (The Recording On event is described in section About Recordable Applications.)

If an application that supports OSA events is launched on a computer for which recording is turned on, the OSA Event Manager will also send it a Recording On event for each recording process that is currently recording.

The recording process receives recordable events by means of a Receive Recordable Event handler-that is, a handler installed in the OSA event dispatch table for event class kCoreEventClass and event ID kAENotifyRecording. Scripting components install this handler on behalf of a recording process when recording is first turned on and remove the handler when recording is turned off. Much like a handler for event class typeWildCard and event ID typeWildCard, the Receive Recordable Event handler handles all recordable events sent to the recording process by the OSA Event Manager. Any other OSA events received by the recording process are dispatched in the usual manner. The Receive Recordable Event handler handles recordable events by recording them in the script specified by the recording process's call to OSAStartRecording.

Receive Recordable Event-receive and record a copy of a recordable event.

Event class	kCoreEventClass
Event ID	kAENotifyRecording
Parameters	Same as OSA event being recorded
Description	Wildcard event class and event ID handled by a recording process in order to receive and record copies of recordable events sent to it by the OSA Event Manager. Scripting components install a handler for this event on behalf of a recording process when recording is turned on and remove the handler when recording is turned off.

Whenever the Receive Recordable Event handler receives a recordable event, the scripting component sends your application a Recorded Text event. The Recorded Text event contains the decompiled source data for the recorded event in the form of styled text. For a description of the Recorded Text event, see Recording Scripts.

When a user turns off recording (for example, by clicking Script Editor's **Stop** button), the recording process calls a scripting component routine (OSAStopRecording) to turn recording off. The scripting component responds by sending a Stop Recording event to the recording process (or any running process on the local computer that has been registered with the OSA Event Manager using the AEInit function).

Stop Recording-stop sending copies of recordable events to recording process

Event class	kCoreEventClass
Event ID	kAEStopRecording
Parameters	None
Description	Sent by a scripting component to the recording process (or to any running process on the local computer that has been registered with the OSA Event Manager using the AEInit function), but handled by the OSA Event Manager. The OSA Event Manager responds by sending a Recording Off event to all running processes on the local computer. This event must be addressed using a process ID (PID); it should never be sent to an address specified as kCurrentProcess.

Like the Start Recording event, the Stop Recording event is handled by the OSA Event Manager. The OSA Event Manager responds by sending a Recording Off event to all running processes on the local computer that have been registered with the OSA Event Manager using the AEInit function. (The Recording Off event is described in section About Recordable Applications.)

Recording continues, and the recording process may continue to receive recordable events, until the OSA Event Manager has notified all running processes that recording has been turned off for that recording process. The OSA Event Manager sends the Recording Off event with the kAEWaitReply flag set to all running processes that have been registered with the OSA Event Manager using the AEInit function. If an application has stored some data (for instance, keystrokes) that needs to be recorded as an OSA event, this is the last chance for the application to send the event for recording purposes. Recording stops only after the OSA Event Manager returns a reply for the Stop Recording event.

The OSA Event Manager supports multiple simultaneous recording processes. A Stop Recording event sent for one of them does not affect the others. If your application needs to know which of several recording processes has turned recording on or off, it can check the keyOriginalAddressAttr attribute of the Recording On or Recording Off event for the address of the recording process.

If the OSA Event Manager does not receive a Stop Recording event for a recording process that quits unexpectedly, the applications being recorded do not find out immediately. When it attempts to send a copy of a recordable event to a recording process that is no longer active, the OSA Event Manager sends a Recording Off event to all running processes that have been registered with the OSA Event Manager using the AEInit function on behalf of that recording process and specifies the address for that process in the keyOriginalAddressAttr attribute. If a recording process that quits is the only actively recording process, recording stops completely after the OSA Event Manager has informed all running processes that have been registered with the OSA Event Manager using the AEInit function that recording has been turned off.

Scripting Components

This chapter describes how your application can use the Component Manager and scripting components to manipulate and execute scripts.

Before you read this chapter, you should read Introduction to Scripting and the chapters about the OSA Event Manager that are relevant to your application.

Your application can use the standard scripting component data structures and routines described in this chapter to manipulate scripts written in any scripting language based on the Open Scripting Architecture (OSA). Your application need not be scriptable or recordable to use these routines. However, if your application is scriptable, you can easily make it capable of manipulating and executing scripts that control its own behavior.

The first section in this chapter describes how to establish a connection with a scripting component. The next two sections provide:

- Examples of how to use the standard scripting component routines
- Information for developers of scripting components

The section Scripting Components Reference describes, in addition to the standard scripting component routines, routines provided by the Object REXX component, routines provided by the generic scripting component, and routines called by scripting components.

Connecting to a Scripting Component

To manipulate and execute scripts written in different scripting languages, your application can use Component Manager methods either to open a connection with each corresponding scripting component individually or to open a single connection with the generic scripting component. The generic scripting component, in turn, attempts to open connections dynamically with the appropriate scripting component for each script. By opening a connection with the generic scripting component, your application can load and execute scripts created by any scripting component that is registered with the Component Manager on the current computer.

In general, you should use the generic scripting component to execute and manipulate existing scripts and a specific scripting component when you create new scripts. When you call OSACompile or OSAStartRecording, the generic scripting component examines the script ID to determine which scripting component to use. If, instead of a script ID, you pass the constant kOSANullScript to these routines, the generic scripting component uses its current *default scripting component* . Each instance of the generic scripting component has its own default scripting component. From the user's point of view, the default scripting component corresponds to the scripting language selected in the Script Editor application when the user first creates a new script.

The generic scripting component provides routines you can use to get and set the default scripting component, determine which scripting component created a particular script, and perform other useful tasks when you are using multiple scripting components. See the section Generic Scripting Component Routines for descriptions of these routines.

You can use the Component Manager method OpenComponent to open a connection to a scripting component you specify with the component identifier returned by the FindNextComponent function. You can also use the OpenDefaultComponent function to open a scripting component without calling the FindNextComponent function.

The OpenComponent and OpenDefaultComponent functions return a component instance. This value identifies your application's connection to a component. You must supply this value whenever you call a standard scripting component method.

Note: Your application may maintain several connections to a single component, or it may have connections to several components at the same time. Because some scripting components can execute only one script at a time per component instance, a multithreaded application

must provide a separate component instance for each script that it compiles or executes while it is simultaneously executing other scripts.

The Component Manager type code for scripting components that support the routines described in this chapter is **osa**, and the subtype code for the generic scripting component is **scpt**.

```
#define kOSAComponentType                          0x2061736F /* "osa " */
#define kOSAGenericScriptingComponentSubtype   0x74706373 /* "scpt" */
```

You can open a connection to a scripting component by calling the OpenDefaultComponent function, which returns a component instance. For example, this code opens a connection with the generic scripting component and stores the returned value in an application-defined variable:

```
/* Get Component Manager */
ComponentManager *cmgr = newComponentManager;

/* open connection to generic scripting component */
OSAScriptingComponent *gScriptingComponent;

gScriptingComponent = cmgr->OpenDefaultComponent(ev, kOSAComponentType,
                      kOSAGenericscriptingComponentSubtype);
```

The generic scripting component in turn opens connections with other scripting components as necessary. The generic scripting component provides routines you can use to get instances of other scripting components when you want to use component-specific routines.

It is also possible to open an explicit connection directly with a specific scripting component such as Object REXX:

```
/* open connection to Object REXX component*/
ComponentInstance gScriptingComponent

gScriptingComponent = cmgr->OpenDefaultComponent(ev, kOSAComponentType,
                                                 kObject REXXSubtype);
```

The scripting component routines described in this chapter include eight groups of optional routines that scripting components can support. If necessary, you can use the FindNextComponent function and other Component Manager routines to find a scripting component that supports a specific group of routines or to determine whether a particular scripting component supports a specific group of routines.

When you call FindNextComponent, you can provide, in a component description record (a data structure of type ComponentDescription), information about the scripting component you wish to find. The flag bits in the *componentFlags* field of a component description record provide this information. To find a scripting component that supports a specific group of optional routines, you can specify one or more of these constants in the *componentFlags* field:

```
#define kOSASupportsCompiling                  0x0002
#define kOSASupportsGetSource                  0x0004
#define kOSASupportsAECoercion                  0x0008
#define kOSASupportsAESending                  0x0010
#define kOSASupportsRecording                  0x0020
#define kOSASupportsConvenience                0x0040
#define kOSASupportsDialects                    0x0080
#define kOSASupportsEventHandling              0x0100
```

The routines that correspond to these constants are described in Optional Scripting Component Routines.

Note: Although the generic scripting component supports all the scripting component routines represented by these flags, the support it can actually provide depends on the individual components with which it opens connections.

The following code fragment shows how you can use these flags and the FindNextComponent function to locate a scripting component with specific characteristics. The *componentFlags* field of the component description record passed to FindNextComponent specifies the flags kOSASupportsCompiling and kOSASupportsGetSource. Because the *componentFlagsMask* field also specifies these flags, the FindNextComponent function locates a scripting component that supports these routines, regardless of whether or not it supports any others. The FindNextComponent function returns a component identifier that you can then use to get more information about the component or to

open it.

```
OSAError MyConnectToScripting(Environment *ev, OSAScriptingComponent **sc)
{
  ComponentDescription descr, descr2;
  Component            comp;
  OSErr                myErr;
  ComponentManager    *cmgr = new ComponentManager;

    /* fill in the fields of the component description record */
    /* first specify component type, subtype, and manufacturer */
    descr.componentType = kOSAComponentType; /* must be scripting component
    descr.componentSubType = (OSType)0;      /* any OSA component matching s
    descr.componentManufacturer = (OSType)0; /* do not care about manufactu

    /* specify component flags and flags mask */
    descr.componentFlags = kOSASupportsCompiling || kOSASupportsGetSource;
    descr.componentFlagsMask = kOSASupportsCompiling || kOSASupportsGetSour

    /* locate and open the specified component */
    myErr = cmgr->FindNextComponent(ev, 0, &descr, &descr2); /* 0 indicates
                                                              /* registered component
                                                              /* will be searched */
    if(myErr)
      return (kComponentNotFound);

    /* check whether the found component is the generic scripting component
    /* if so, skip it and find the next matching component */
    if(descr2.componentSubType == kOSAGenericScriptingComponentSubtype)
      myErr = cmgr->FindNextComponent(ev, &descr2, &descr, &descr2);

    if(myErr)
      return (kComponentNotFound);
    else
    {
      *sc = (OSAScriptingComponent *) cmgr->OpenComponent(ev, &descr2);
      if(*sc == 0)
        return (kComponentNotFound);
      else
        return (noErr);

    }
}
```

Because the generic scripting component supports all the standard scripting component routines, the MyConnectToScripting function in the previous code fragment checks whether the found component is the generic scripting component and, if so, skips it. If for any reason FindNextComponent cannot locate and open a scripting component that supports the specified routines, MyConnectToScripting returns the application-defined constant kComponentNotFound.

Using Scripting Component Routines

The following sections describe how to use some of the standard scripting component routines to manipulate and execute scripts from within your application. For an overview of these routines, see Manipulating and Executing Scripts.

The first section describes how to compile and execute source data for a script. The remaining sections describe how you can use scripting component routines to:

- Get a handle to a compiled script and save the data as a resource
- Load and execute a previously saved and compiled script
- Load, modify, recompile, and save a compiled script
- Redirect OSA events to handlers in script contexts
- Supply a resume dispatch function
- Supply an alternative active function
- Supply alternative send and create functions
- Record OSA events in compiled scripts and display equivalent source data to the user

---

Compiling and Executing Source Data

This section describes how you can use scripting component routines to obtain source data from users, compile the source data, and execute the compiled script. To create and execute a script using the Script Editor application, a user can type the script, then click the **Run** button to execute it. Your application can provide similar capabilities.

To allow users to write a new script and then execute it, your application must use scripting component routines to compile and execute the source data. To compile source data in a new script with a new script ID, pass the constant kOSANullScript (rather than an existing script ID) in the *ppreviousAndResultingScriptID* parameter of the OSACompile function. This causes OSACompile to return a new script ID in the same parameter.

To execute a compiled script, your application must specify, in addition to the script ID for the compiled script, a script context: either the corresponding scripting component's default context or a script ID for the global context created by that scripting component. Script contexts maintain state information for the execution of scripts. Your application can use script contexts to control the binding of variables used in scripts that it executes. For example, if your application saves its own global context and reuses it every time a script is executed, the binding of variables used in the script is maintained after the user restarts the computer. If your application does not specify a script context, the Object REXX component uses a single default context whenever it executes the script. A scripting component's default context binds the variables used in the script only until the user quits the application.

To specify a scripting component's default context, pass the constant kOSANullScript in the *contextID* parameter of the OSAExecute function; to specify some other global context, pass its script ID in the *contextID* parameter.

The MyDoNewScript procedure in the following code fragment allows a user to type a script in the appropriate scripting language, then compiles the script, executes the compiled script using a global context provided by the application, and displays the result to the user.

The MyDoNewScript procedure begins by calling the OSAScriptingComponentName function to obtain the name of the scripting component specified by the *presultingScriptingComponentName* parameter. This name is passed to the application-defined function MyGetUserScriptText.

Note: If you are using the generic scripting component, you can use the OSAGetDefaultScriptingComponent function to get the subtype code for the default scripting component (that is, the scripting component used by the generic scripting component for new scripts). You can then get an instance of the default scripting component by passing its subtype code to OSAGetScriptingComponent. Finally, you can pass that instance to OSAScriptingComponentName to obtain the default scripting component's name. For more information about the default scripting component and routines you can use with the generic scripting component, see Generic Scripting Component Routines.

The MyGetUserScriptText function displays the name of the scripting language to use in a script-editing window or message box that allows the user to type and execute a new script. After it obtains the source data for the new script, the MyDoNewScript procedure sets the *scriptID* parameter to kOSANullScript. The procedure then passes the source data and script ID to the OSACompile function. When the script ID passed to OSACompile is kOSANullScript, OSACompile returns, in the same parameter, a new script ID for the resulting compiled script. The MyDoNewScript procedure then passes the new script ID to the OSAExecute function.

In addition to a component instance and the script ID for the compiled script to be executed, OSAExecute takes a script ID for a script context and a parameter that contains the mode flags, if any, for script execution. In the following code fragment, the script ID passed to OSAExecute for the script context is gContext, a global context provided by the application. The constant kOSAModeNull in the next parameter indicates that no mode flags are set for script execution.

```
OSAError MyDoNewScript (Environment *ev, OSAScriptingComponent *sc)
{
  AEDesc     componentName, scriptText, resultText;
  OSAID       scriptID, resultID;
  OSAError   myOSAErr, ignoreErr;

  /* get the scripting component's name so you can show */
  /* the user which scripting language to use */
```

```
    myOSAErr = sc->OSAScriptingComponentName(ev, componentName);
    if (myOSAErr == noErr) {
      /* get the user's script text, then compile it */
      MyGetUserScriptText(componentName, scriptText);

      /* to create a new compiled script using the user's script */
      /* text, pass kOSANullScript to OSACompile as the script ID */
      /* for the script to be compiled */
      scriptID = kOSANullScript;
      myOSAErr = sc->OSACompile(ev, &scriptText, kOSAModeNull,
                                scriptID);
      ignoreErr = AEDisposeDesc(&scriptText);
  }
  if (myOSAErr == noErr) {
      /* execute the script in a global context */
      myOSAErr = sc->OSAExecute(ev, scriptID, gContext, kOSAModeNull,
                                resultID);
      ignoreErr = sc->OSADispose(ev, scriptID);
      if (myOSAErr == noErr) {
        /* convert the script value returned by OSAExecute to */
        /* text that can be displayed to the user */
        myOSAErr = sc->OSADisplay(ev, resultID, typeChar, kOSAModeNull,
                                  resultText);
        ignoreErr = sc->OSADispose(ev, resultID);
        /* show result to user */
        MyShowUserResult(resultText);
        ignoreErr = AEDisposeDesc(&resultText);
      }
  }
  if (myOSAErr == errOSAScriptError)
      MyGetScriptErrorInfo();
}
```

If script execution is successful, the MyDoNewScript procedure passes the script ID for the resulting script value to the OSADisplay function and calls the MyShowUserResult procedure to display the script value to the user. It also disposes of the script data for the compiled script. If OSAExecute or OSACompile returns the result code errOSAScriptError, the MyDoNewScript procedure calls the MyGetScriptErrorInfo procedure shown in the following code fragment, which uses the OSAScriptError function to obtain more information about the error.

Whenever a scripting component routine returns the result code errOSAScriptError, you can use OSAScriptError to obtain more information about the error. The *selector* parameter of the OSAScriptError function is a constant that specifies the kind of error information to be returned, and the *desiredType* parameter is the descriptor type for the descriptor record in which the additional error information will be returned.

The MyGetScriptErrorInfo procedure in the following code fragment calls OSAScriptError three times: once to obtain an error number for either a system error or a scripting component error, once to obtain a text description of the error, and once to obtain error-range information. Finally, the MyGetScriptErrorInfo procedure extracts the starting and ending positions of the error range in the source data and calls the application-defined procedure MyIndicateError to display the error information to the user. Note that your application is responsible for disposing of any descriptor records that are created.

You should use the OSACompile and OSAExecute functions, as shown in the previous code fragment,, if you expect the user to execute the compiled script several times or manipulate it in some other way. If you want to compile and execute a script just one time and do not need to keep the compiled script in memory after it has been executed, you can use either OSACompileExecute or OSADoScript if these functions are supported by the scripting component you specify.

The OSACompileExecute function takes a component instance, a descriptor record for the source data to be compiled and executed, a context ID, and a mode flags parameter. It executes the resulting compiled script, disposes of the compiled script, and returns the script ID for the resulting script value.

The OSADoScript function takes a component instance, a descriptor record for source data, a context ID, a text descriptor type, and a mode flags parameter. It compiles and executes the script, returns a descriptor record for the text that corresponds to the resulting script value, and disposes of both the compiled script and the script value.

```
OSAError MyGetScriptErrorInfo (Environment *ev, OSAScriptingComponent *sc)
{
```

```
Handle       errorMessage;
INT          startPos, endPos;
AEDesc       desc, recordDesc;
DescType     actualType;
Size         actualSize;
Ptr          scriptErr;
OSErr        myErr, ignoreErr;
OSAError     myOSAErr;

  myOSAErr = sc->OSAScriptError(ev, kOSAErrorNumber, typeShortInteger, desc
  myErr = AEGetDescData(&desc, &actualType, scriptErr,
                        sizeof(scriptErr), &actualSize);
  ignoreErr = AEDisposeDesc(&desc);
  myOSAErr = sc->OSAScriptError(ev, kOSAErrorMessage, typeChar, desc);
  errorMessage = desc.dataHandle;
  myOSAErr = sc->OSAScriptError(ev, kOSAErrorRange, typeOSAErrorRange, desc
  myErr = AECoerceDesc (&desc, typeAERecord, &recordDesc);
  ignoreErr = AEDisposeDesc(&desc);
  myErr = AEGetKeyPtr(&recordDesc, keySourceStart, typeShortInteger,
                      &actualType, &startPos, sizeof(startPos),
                      &actualSize);
  myErr = AEGetKeyPtr(&recordDesc, keySourceEnd, typeShortInteger,
                      &actualType, &endPos, sizeof(endPos),
                      &actualSize);
  ignoreErr = AEDisposeDesc(&recordDesc);
  MyIndicateError(scriptErr, errorMessage, startPos, endPos);
  /* add your own error checking */
}
```

Saving Script Data

After creating a new script (or after modifying a previously saved script), a user may want to save it.

Important: Your application should usually save scripts as script data rather than source data, so that it can reload and execute the data without compiling it.

Before saving script data, your application can use the OSAStore function to obtain a handle to the data. The OSAStore function takes four input parameters: a component instance that identifies a connection with a scripting component, a script ID for the script data to be stored, a desired descriptor type for the descriptor record to be returned, and a parameter that contains mode flags for use by individual scripting components. It returns a descriptor record for the script data in the *presultingScriptData* parameter.

The sections that follow describe the storage formats used by OSAStore and the resource and file types for script data.

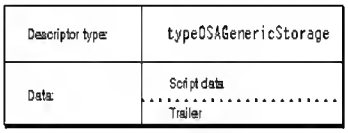
Storage Formats for Script Data

The descriptor record returned by OSAStore can be either a generic storage descriptor record or a component-specific storage descriptor record:

- A *generic storage descriptor record* is a special kind of descriptor record of type typeOSAGenericStorage that can be used to store script data created by any scripting component.
- A *component-specific storage descriptor record* is a descriptor record whose descriptor type is the scripting component subtype

value for the scripting component that created the script data.

The following figure illustrates the logical arrangement of a generic storage descriptor record. The descriptor type for a generic storage descriptor record is always typeOSAGenericStorage, and the data referred to by the descriptor record's handle is always followed by a trailer containing the subtype value for the scripting component that created the script data.



The following figure illustrates the logical arrangement of a component-specific storage descriptor record. The descriptor type for a component-specific storage descriptor record is the subtype value for the scripting component that created the script data, and the data referred to by the descriptor record's handle consists of the script data only, with no trailer.

Descriptor type:	Scripting component subtype (for example, typeDREXXtorage
Data:	Script data

In most cases, it is safest to request a handle to script data in the form of a generic storage descriptor record, regardless of the scripting component subtype you pass to the OSAStore function.

If the presence of the trailer in a generic storage descriptor record does not interfere with the script data, that data may be used for a wide variety of purposes. However, in some cases adding a trailer to script data may interfere with script execution. For example, suppose the data for a generic storage descriptor record consists of sound data. If a scripting component attempts to play the data from beginning to end as sound data, the trailer will interfere with the resulting sound. In this case, an application must open an explicit connection with a scripting component that can play sounds before saving the data, and then call OSAStore with a desired type that consists of the subtype for that scripting component.

Loading and Executing Script Data

The figure section Scripting Components and Applications That Execute Scripts illustrates how an application might execute a script whenever the user presses the **Tab** key after entering a customer's name in the "Customer Name" field of an electronic form. To execute a script in response to some user action, your application must be able to load and execute the script data for a compiled script.

This section describes how to load and execute a previously compiled and saved script. The next section describes how to allow a user to modify a compiled script.

The OSALoad function takes two input parameters: a descriptor record that contains a handle to the script data to be loaded; and a parameter that contains flags for use by individual scripting components. The function returns, in the *presultingScriptID* parameter, a script ID for the script data.

When your application calls OSALoad with a component instance that identifies a connection with the generic scripting component, the generic scripting component in turn uses a connection with the scripting component that created the script data (if that component is registered with the Component Manager on the local computer). If the descriptor record passed to OSALoad is of type typeOSAGenericStorage, the generic scripting component uses the trailer that follows the script data to determine which scripting component to open a connection with. If the descriptor record's type is the subtype value for some other scripting component, the generic scripting component does not look for a trailer and uses the descriptor type to identify the scripting component.

When your application calls OSALoad with a component instance that identifies a connection to any scripting component other than the generic scripting component, that component can load script data only if it was saved as the data for a descriptor record whose descriptor type matches the scripting component's subtype. In this case, however, your application easily can take advantage of additional routines and other special capabilities provided by that scripting component.

It is also possible to call OSALoad using the generic scripting component, then use generic scripting component routines to identify the specific component associated with the loaded script. This allows you to use component-specific routines with a script originally loaded by the generic scripting component. For information about how to do this, see Routines Used by Scripting Components.

The OSALoad function returns a script ID for the loaded script data. The generic scripting component always associates the returned script ID with the scripting component that created the script. In this way, it can use a connection with that component again whenever the client application passes the returned script ID to other scripting component routines.

The following code fragment shows a procedure that loads and executes a script. The MyLoadAndExecute procedure takes a handle to script data that was previously saved using a generic storage descriptor record, obtains a script ID for the equivalent compiled script, executes the

compiled script in the default context, and disposes of both the compiled script and the resulting script value ID. If the OSAExecute function returns a script execution error, MyLoadAndExecute obtains further information about the error and displays it to the user.

```
OSAError MyLoadAndExecute (Environment *ev, OSAScriptingComponent *sc,
                           PBYTE scriptData, ULONG size)
{
  AEDesc     scriptDesc;
  OSAID     scriptID, resultID;
  AEDesc     scriptText;
  OSAError   myOSAErr, ignoreErr;

  /*load the script data*/
  MyInitScriptDesc(&.ScriptDesc, scriptData, size);
  myOSAErr = sc->OSALoad(ev, scriptDesc, kOSAModeNull, scriptID);
  if (myOSAErr == noErr) {
      /*execute the resulting compiled script in the default */
      /* context*/
      myOSAErr = sc->OSAExecute(ev, scriptID, kOSANullScript, kOSAModeNull,
                                resultID);
      ignoreErr = sc->OSADispose(ev, scriptID);
      ignoreErr = sc->OSADispose(ev, resultID);
  }
  if (myOSAErr == errOSAScriptError)
      MyGetScriptErrorInfo;
}
```

The OSALoad function in the previous code fragment takes a component instance, a generic storage descriptor record for the script data to be loaded, and a parameter that contains the mode flags, if any, for loading the script. In this case the constant kOSAModeNull indicates that no mode flags are set. The OSALoad function returns a script ID for the resulting compiled script, which the MyLoadAndExecute procedure then passes to the OSAExecute formt=extonly function.

In addition to a component instance and the script ID for the compiled script to be executed, the OSAExecute function takes a script ID for a context and a parameter that contains the mode flags, if any, for script execution. In the previous code fragment, the script ID passed to OSAExecute for the script context is kOSANullScript, indicating that the scripting component can use its default context to bind any variables. The constant kOSAModeNull in the *modeFlags* parameter indicates that no mode flags are set for script execution.

After disposing of the compiled script and the resulting script value, MyLoadAndExecute checks the result code returned by OSAExecute. If it is errOSAScriptError, MyLoadAndExecute calls the MyGetScriptErrorInfo procedure (see the second code fragment in section Compiling and Executing Source Data), which in turn uses the OSAScriptError function to obtain more information about the error.

You can use the OSAExecute and OSAExecute functions as shown in the previous code fragment if you expect the user to execute the compiled script several times or manipulate it in some other way. If you want to load and execute a script just one time and do not need to keep the compiled script in memory after it has been executed, you can use OSALoadExecute instead of OSALoad, OSAExecute, and OSADispose. This function takes a component instance, a descriptor record for the script data to be loaded and executed, a context ID, and a mode flags parameter. The OSALoadExecute function executes the resulting compiled script, disposes of the compiled script, and returns the script ID for the resulting script data.

Modifying and Recompiling a Compiled Script

In addition to loading and executing a previously compiled and saved script as described in the previous section, your application can use the scripting component routines described in this section to decompile a compiled script, display the equivalent source data to users for editing, and recompile the source data after editing is completed. For example, if a user wants to change the script shown in the figure in section Scripting Components and Applications That Execute Scripts so that it refers to some other database or looks up other information in addition to the customer's address, the forms application can use scripting component routines to display the compiled script to the user and recompile it after the user has modified it.

You can use the OSAGetSource function to obtain the source data for a compiled script. The OSAGetSource function takes a component instance, a script ID for the compiled script, and the desired type of the resulting descriptor record. If you specify a component instance that identifies a connection with the generic scripting component, you can use OSAGetSource to get the source data for any compiled script created by a scripting component that is registered with the Component Manager on the local computer. If you specify a component instance that identifies an explicit connection with a scripting component, you can use OSAGetSource only to get the source data for scripts that were compiled by that scripting component.

The MyEditGenericScript procedure in the following code fragment shows how you can use the OSAGetSource function with a component instance that identifies a connection to the generic scripting component. The MyEditGenericScript function gets the source data for a compiled script, allows the user to edit it, and recompiles the script so the original script ID refers to the recompiled script data.

```
OSAError MyEditGenericScript (Environment *ev, OSAScriptingComponent *sc,
                              OSAID scriptID)
{
AEDesc       scriptText;
OSAError     myOSAErr;
OSErr       ignoreErr;

  /* first get the source data */
  myOSAErr = sc->OSAGetSource(ev, scriptID, typeChar, scriptText);

  /* call the application's primitive text editor */
  MyEditText(&scriptText);

  /*now compile the edited script data in scriptText using */
  /* the scripting component that originally created it; */
  /* passing the original script ID to OSACompile causes */
  /* OSACompile to replace the original script with the new one*/
  myOSAErr = sc->OSACompile(ev, scriptText, kOSAModeNull, scriptID);
  ignoreErr = AEDisposeDesc(&scriptText);
  if (myOSAErr == errOSAScriptError)
      MyGetScriptErrorInfo;
}
```

After obtaining the source data for the script, the MyEditGenericScript procedure calls the MyEditText function, which displays the application's own primitive text editor and allows the user to edit the source data. After the user has finished editing the script, MyEditGenericScript passes the edited text and the script ID for the original compiled script to the OSACompile function, which updates the script ID so that it refers to the modified and recompiled script. The kOSAModeNull constant passed in the *modeFlags* parameter of OSACompile indicates that no mode flags are specified for compilation.

If the OSACompile function returns errOSAScriptError, the MyEditGenericScript procedure calls the MyGetScriptErrorInfo procedure shown in the second code fragment in section Compiling and Executing Source Data to obtain information about the error.

After script data has changed as shown in the previous code fragment, your application should save the modified script data. The following code fragment shows how this could be done from a function that loads script data, calls the MyEditGenericScript procedure shown in the previous code fragment to modify and recompile the script, then saves the modified script data.

```
OSAError MyLoadAndModifyScriptData(Environment *ev, OSAScriptingComponent *
                                   PBYTE scriptData, ULONG size)
{
  AEDesc   scriptDesc, storageDescRec;
  OSAID   scriptID;
  OSAError myOSAErr;
  OSErr   ignoreErr;

    AECreateDesc(typeOSAGenericStorage, scriptData, size, &scriptDesc);
    myOSAErr = gsc->OSALoad(ev, &scriptDesc, kOSAModeNull, scriptID);
    MyEditGenericScript(scriptID);
    myOSAErr = gsc->OSAStore(ev, scriptID, typeOSAGenericStorage,
                             kOSAModeNull, &torageDescRec);
    MySaveScriptData(&storageDescRec);
    ignoreErr = AEDisposeDesc(&scriptDesc);
    ignoreErr = AEDisposeDesc(&storageDescRec);
}
```

Using a Script Context to Handle an OSA Event

The preceding sections describe how you can load, compile, modify, and execute scripts under circumstances determined by your application. Your application can use these techniques to associate a script with an OSA event object or application object and execute the script when the user manipulates the object in some way.

Another way to execute a script is to use a script context (called a *script object* in Object REXX) to handle an OSA event. To do this, your application passes both the event and the script context to OSAExecuteEvent or OSADoEvent. You can also associate script contexts with OSA event objects-that is, objects in your application that can be identified by object specifier records. If an OSA event acts on an object with which a script context is associated, your application attempts to use the script context to handle the OSA event.

For example, the figure in section Using a Script Context to Handle an OSA Event shows how you can use a general OSA event handler to provide initial processing for all OSA events received by your application. the following code fragment shows an example of such a handler.

You install a general OSA event handler like the one in the following code fragment in your application's special handler dispatch table using the constant keyPreDispatch:

```
myErr = AEInstallSpecialHandler(keyPreDispatch, &MyGeneralOSAEventHandler,
                                FALSE);
```

When it receives an OSA event, the MyGeneralOSAEventHandler function in the following code fragment first extracts the event's direct parameter. It then calls another application-defined function, MyGetAttachedScript, which checks whether the direct parameter contains an object specifier record, calls AEResolve to locate the corresponding OSA event object, and returns a script ID for any script context attached to that object.

If a script context is associated with the object, MyGeneralOSAEventHandler passes the script context's script ID and the OSA event to the OSADoEvent function. Otherwise, MyGeneralOSAEventHandler returns errAEEventNotHandled, which causes the OSA Event Manager to look for an appropriate handler in the application's OSA event dispatch table or elsewhere using standard OSA event dispatching.

The OSADoEvent function in the following code fragment takes a component instance that identifies a connection with the generic scripting component. (If it has not already done so, the generic scripting component in turn opens a connection with the scripting component that created the script context.) In addition to the component instance, the OSA event, and the script ID for the script context, OSADoEvent takes a parameter that indicates no mode flags are set and a VAR parameter that contains any reply OSA event returned as a result of handling the event.

If the scripting component determines that a script context cannot handle the specified event (for example, if an Object REXX script context does not include statements that handle the event), OSADoEvent returns errAEEventNotHandled. If OSADoEvent attempts to use the script context to handle the event, the function returns a reply event that contains either the resulting script value or, if an error occurred, information about the error.

The script context shown in the figure in section Using a Script Context to Handle an OSA Event contains an Object REXX handler for the Move event. Such handlers exist only as Object REXX statements in a script context and do not have corresponding entries in an application's OSA event dispatch table. However, a handler in a script context can modify or override the actions performed by an application's standard OSA event handlers installed in its OSA event dispatch table.

```
OSErr MyGeneralOSAEventHandler (Environment *ev, OSAEvent event, OSAEvent r
                                long refcon)
{
AEDesc     dp, resultDesc;
OSAID       scriptID;
OSErr       myErr, ignoreErr;
OSAError   myOSAErr;

  /* get the direct parameter */
  myErr = AEGetParamDesc(&event, keyDirectObject, typeWildCard, &dp);
  /*get script ID for script context attached to object */
  /* specified in direct parameter*/
  if (MyGetAttachedScript(dp, scriptID)) {
      /*execute the handler in the script context handler and, if */
      /* necessary, the default OSA event handler*/
      myOSAErr = sc->OSADoEvent(ev, event, scriptID, kOSAModeNull, reply);
```

```
  }
  myOSAErr = errAEEventNotHandled;
  ignoreErr = AEDisposeDesc(&dp);
  return ((OSErr)myOSAErr);
}
```

Supplying Alternative Create and Send Functions

Every scripting component calls a *create function* whenever it creates an OSA event during script execution, and a *send function* whenever it sends an OSA event. Scripting components that use OSA events during script compilation, including Object REXX, also call create and send functions during compilation.

Some scripting components may provide routines that allow your application to set or get the pointers to the create and send functions used by that scripting component. If your application does not set alternative send and create functions, the scripting component uses the standard OSA Event Manager functions AESend and AECreateOSAEvent which it calls with its own default parameters.

A scripting component that supports the routines you can use to set or get alternative create and send functions has the kOSASupportsAESending bit set in its component description record. For more information about using the Component Manager to find a scripting component that supports specific routines, see Connecting to a Scripting Component.

Alternative Create Functions

A scripting component that allows your application to set or get its create function uses a pointer to identify the current create function.

```
TYPE OSAActiveProcPtr = ProcPtr;
```

A pointer of type AECreateOSAEventUPP points to a MyAECreateProc function that takes the same parameters as the AECreate function plus a reference constant.

```
OSErr MyAECreateProc (AEEventClass theAEEventClass, AEEventID theAEEventID,
                      AEAddressDesc target, int returnID, long transactionI
                      OSAEvent result, long refCon);
```

Your application can use an alternative create function to gain control over the creation and addressing of OSA events. This can be useful, for example, if your application needs to add its own transaction code to the event.

To set an alternative create function, call OSASetCreateProc; to get the current create function, call OSAGetCreateProc. If you do not set an alternative create function for a scripting component, it uses the standard OSA Event Manager function AECreateOSAEvent, which it calls with its own default parameters.

Your alternative create function can, in turn, call the scripting component's default create function. To do this, your application can call OSAGetCreateProc before calling OSASetCreateProc to set the alternative create function and then call the default create function directly when necessary.

Alternative Send Functions

A scripting component that allows your application to set or get its send function uses a pointer to identify the current send function.

```
TYPE AESendProcPtr = ProcPtr;
```

A pointer of type AESendProcPtr points to a MyAESendProc function that takes the same parameters as the AECreate function plus a reference constant.

```
OSErr MyAESendProc (OSAEvent theOSAEvent, OSAEvent reply, AESendMode sendMc
                    AESendPriority sendPriority, long timeOutInTicks,
                    IdleProcPtr idleProc, EventFilterProcPtr filterProc,
                    long refCon);
```

Your application can use an alternative send function to perform almost any action instead of or in addition to sending OSA events. For example, it can modify OSA events before sending them, save copies of OSA events before sending them, or substitute some other specialized mechanism for sending OSA events.

To set an alternative send function, call OSASetSendProc; to get the current send function, call OSAGetSendProc. If you do not set an alternative send function for a scripting component, it uses the standard OSA Event Manager function AESend, which it calls with its own default parameters.

Your alternative send function can, in turn, call the scripting component's default send function. To do this, your application can call OSAGetSendProc before calling OSASetSendProc to set the alternative send function and then call the default send function directly when necessary.

After a scripting component successfully calls a send function, the scripting component proceeds with script execution. If a call to a send function is not successful, the scripting component returns errOSAScriptError, and a subsequent call to OSAScriptError with kOSAErrorNumber in the *selector* parameter returns errAEEventNotHandled.

Multithreaded applications need to allow other threads to execute while one thread is waiting for the response to an OSA event. You can accomplish this by supplying an idle function for your alternative send function that allows threads to be switched and by setting the kAEQueueReply flag in the *sendMode* parameter of the send function; however, if the call to the send function specifies the kAENoReply flag, be careful not to override it, because the user may have explicitly requested that no reply be returned or the **aete** resource may indicate that the application cannot reply to that event.

Some scripting components can execute only one script at a time per component instance. For this reason, a multithreaded application must provide a separate component instance for each script that it compiles or executes while it is also compiling or executing other scripts.

You should follow the rules for setting send mode flags described in Creating and Sending OSA Events when you set flags for the *sendMode* parameter of an alternative send function. Keep these additional guidelines in mind:

- If the target application is on the local computer, you can set the kAECanInteract and kAECanSwitchLayer flags.
- If the target application is on the local computer and the user has requested no reply, set the kAENoReply, kAECanInteract, and kAECanSwitchLayer flags.
- If the target application is on a remote computer, set the kAENeverInteract flag and do not set the kAECanSwitchLayer flag.

Recording Scripts

If you want your application to record OSA events in the form of a compiled script, or if you are writing a script-editing application like Script Editor, you can use the OSAStartRecording and OSAStopRecording functions to start and stop recording into a specified script ID on a single computer. Both functions take a component instance and a script ID for a compiled script. When your application calls OSAStartRecording the scripting component identified by the component instance sends a Start Recording event to your application and installs a Receive Recordable Event handler in your application's OSA event dispatch table. When your application calls OSAStopRecording the scripting component removes the handler.

An application acting as a recording process in this manner should not provide a handler for the Start Recording event. Instead, the OSA Event Manager receives the event and responds by sending a Recording On event to all running processes on the local computer. Thereafter, the OSA Event Manager sends copies of subsequent recordable events to the recording process, whose previously installed Receive Recordable Event handler, much like a handler for event class typeWildCard and event ID typeWildCard, handles those recordable events by recording them in the compiled script specified in the call to OSAStartRecording.

Whenever the Receive Recordable Event handler receives a recordable event, the scripting component sends your application a Recorded Text event. The Recorded Text event contains the decompiled source data for the recorded event in the form of styled text.

Recorded Text-append styled text to script editor window.

Event class	kOSASuite

Event ID	kOSARecordedText
Required parameter	
Keyword:	keyDirectObject
Descriptor type:	typeStyledText or any other text descriptor type
Data:	Decompiled source data for recorded event
Description	Sent by a scripting component to a recording process for each event recorded after a call to OSAStartRecording

If your want your application to display the source data for recorded events as they are recorded, you must provide a handler for the Recorded Text event.

For more information about the Receive Recordable Event handler and OSA event recording, see How OSA Event Recording Works.

Writing a Scripting Component

It is possible to create scripting components that execute a variety of scripts, including scripts that can be "run" in some sense but do not consist of statements in a scripting language. For example, script data can consist of an XCMD or even sound data that the appropriate scripting component can trigger or play back when it executes the script (see Storage Formats for Script Data).

Every scripting component should also:

- Provide a component name in the scripting component's component resource that will make sense when displayed to users.
- Support the standard scripting component routines described in Required Scripting Component Routines.
- Support some, all, or none of the optional scripting component routines, as appropriate for the tasks to be performed by the scripting component. These routines are described in Optional Scripting Component Routines.
- Use the three OSA routines OSAGetStorageType, OSAAddStorageType, and OSARemoveStorageType to inspect, add, or remove the trailers appended to script data in generic storage descriptor records. These routines are described in Manipulating Trailers for Generic Storage Descriptor Records.
- Send the Get AETE event when necessary. This event is described in Handling the Get AETE Event.

Scripting Components Reference

This section describes the standard scripting component data structures and routines your application can use to manipulate and execute scripts. This section also describes additional routines provided by the Object REXX scripting component and three routines called by scripting components.

The first section, Data Structures, describes the principal data types used by scripting component routines. Required Scripting Component Routines describes the standard scripting component routines that all scripting components must support. Optional Scripting Component Routines describes additional standard scripting component routines that scripting components are not required to support.

Your application can use the Component Manager to find a scripting component that supports specific optional routines or to determine whether a particular scripting component supports a specific group of routines. For information about how to do this, see Connecting to a Scripting Component.

Generic Scripting Component Routines describes routines you can use to get instances of specific components and perform other useful tasks when you are using multiple scripting components. Routines Used by Scripting Components describes three routines that all scripting components can use to manipulate trailers for generic storage descriptor records.

Data Structures

This section describes the principal data structures and Component Manager type codes used by the standard scripting component routines.

Data structures used by individual routines are described with the appropriate routines in the sections that follow.

The Component Manager type code for components that support the standard scripting component routines is **osa** , and the subtype code for the generic scripting component is **scpt**.

```
#define kOSAComponentType                          0x2061736F /* "osa " */
#define kOSAGenericScriptingComponentSubtype   0x74706373 /* "scpt" */
```

Because all results returned by the Component Manager are of type OSErr (a long integer), scripting components also define this type for result codes.

```
typedef LONG OSErr;
```

Scripting components keep track of script data in memory by means of script IDs of type OSAID.

```
typedef unsigned long OSAID;
```

A scripting component assigns a script ID when it creates the associated script data (that is, a compiled script, a script value, a script context, or other kinds of script data supported by a scripting component) or loads it into memory. The scripting routines that create, load, compile, and execute scripts all return script IDs, and your application must pass valid script IDs to the other routines that manipulate scripts. A script ID remains valid until a client application calls OSADispose to reclaim the memory used for the corresponding script data.

If the execution of a script does not result in a value, OSAExecute returns the constant kOSANullScript as the script ID. If a client application passes kOSANullScript to the OSAGetSource function instead of a valid script ID, the scripting component should display a null source description (possibly an empty text string). If a client application passes kOSANullScript to OSAStartRecording, the scripting component creates a new compiled script for editing or recording.

```
#define kOSANullScript ((OSAID) 0)
```

Required Scripting Component Routines

This section describes the routines that all scripting components must support. Your application can use these routines to save and load script data, execute and dispose of scripts, get script information, and manipulate the active function. Optional Scripting Component Routines describes additional routines your application can use with scripting components that support them.

Saving and Loading Script Data

The OSAStore function takes a script ID and returns a copy of the corresponding script data in the form of a storage descriptor record. You can then save the script data as a resource or write it to the data fork of a document. The OSALoad function takes script data in a storage descriptor record and returns a script ID.

Executing and Disposing of Scripts

To execute a script, your application must first obtain a valid script ID for a compiled script or script context. You can use either the OSALoad function described in the preceding section or the optional OSACompile function to obtain a script ID.

The OSAExecute function takes a script ID for a compiled script or script context and returns a script ID for a script value. The OSADisplay function converts a script value to text that your application can later display to the user. If the OSAExecute function returns errOSAScriptError, you can use the OSAScriptError function to get more information about the error.

When your application no longer needs the script data associated with a specific script ID, you can use the OSADispose function to release the memory the script data occupies.

Setting and Getting Script Information

The OSASetScriptInfo function sets various kinds of information about script data, and the OSAGetScriptInfo function returns information about script data. The kind of information these functions set or get depends on constants you pass to the functions.

Manipulating the Active Function

The OSASetActiveProc and OSAGetActiveProc functions allow your application to set or to get a pointer to the active function called periodically by the scripting component during script execution. To get time periodically during script execution for its own purposes, your application can substitute its own active function for use by the scripting component. If you do not specify an active function, the scripting component uses its default active function, which allows a user to cancel script execution.

The functions described in this section use the following type for pointers to active functions:

```
TYPE OSAActiveProcPtr = ProcPtr;
```

Optional Scripting Component Routines

This section describes eight groups of optional routines that scripting components can support. Your application can use the Component Manager to find a scripting component that supports a specific group of routines or to determine whether a particular scripting component supports a specific group of routines.

To specify one or more groups of routines for the Component Manager, use the following constants to set the equivalent bits in the componentFlags field of a component description record:

```
#define kOSASupportsCompiling               0x0002
#define kOSASupportsGetSource               0x0004
#define kOSASupportsAECoercion               0x0008
#define kOSASupportsAESending               0x0010
#define kOSASupportsRecording               0x0020
#define kOSASupportsConvenience             0x0040
#define kOSASupportsDialects                 0x0080
#define kOSASupportsEventHandling           0x0100
```

Each of these flags identifies one of the groups of routines that are described in the sections that follow. For information about using these constants to locate scripting components that support specific groups of optional routines, see Connecting to a Scripting Component.

Compiling Scripts

Scripting components can provide three optional routines that get the name of a scripting component, compile a script, and update a script ID.

To obtain the name of a scripting component in a form that you can coerce to text, you can use the OSAScriptingComponentName function.

The OSACompile function compiles source data and returns a script ID, and the OSACopyID function updates the script data associated with one script ID with the script data associated with another script ID.

A scripting component that supports the routines in this section has the kOSASupportsCompiling bit set in the *componentFlags* field of its component description record.

Getting Source Data

The OSAGetSource function returns the source data that corresponds to the script data identified by a script ID. The source data it returns can, in turn, be passed to OSACompile.

A scripting component that supports the OSAGetSource function has the kOSASupportsGetSource bit set in the *componentFlags* field of its component description record.

Coercing Script Values

Scripting components can provide support for two optional routines, OSACoerceFromDesc and OSACoerceToDesc, which coerce data in a descriptor record to a script value and coerce a script value to data in a descriptor record, respectively.

A scripting component that supports the routines in this section has the kOSASupportsAECoercion bit set in the *componentFlags* field of its component description record.

Manipulating the Create and Send Functions

Some scripting components provide routines that allow your application to set or get pointers to the create and send functions used by the scripting component when it sends and creates OSA events during script execution. If you do not set the pointers that specify these functions, the scripting component uses the standard AECreateOSAEvent and AESend functions with default parameters.

To gain control over the creation and addressing of OSA events, your application can provide its own create function for use by scripting components. To set a new create function, call the OSASetCreateProc function; to get the current create function, call OSAGetCreateProc.

The send function provided by your application can perform almost any action instead of or in addition to sending OSA events; for example, it can be used to facilitate concurrent script execution. To set a new send function, call the OSASetSendProc function; to get the current send function, call OSAGetSendProc.

The functions described in this section use the following types for pointers to the create and send functions:

```
TYPE
        AESendProcPtr = ProcPtr;
        AECreateOSAEventProcPtr = ProcPtr;
```

For more information about create and send functions, see Supplying Alternative Create and Send Functions.

Scripting components that support manipulation of the create and send functions also support the OSASetDefaultTarget function, which allows you to set the default application to which OSA event are sent.

A scripting component that supports the functions described in this section has the kOSASupportsAESending bit set in the *componentFlags* field of its component description record.

Recording Scripts

The OSAStartRecording function turns on the OSA Event Manager's recording mechanism and specifies a script in which subsequent

recordable OSA events are recorded. The scripting component sends the recording process (for example, a script editor) a Recorded Text event that contains the decompiled equivalent for each recordable event it receives. The script editor can then display the decompiled script in a script editor window if a window for that script is currently open. Recording continues until a call to OSAStopRecording turns recording off.

Script editors use these routines to allow users to control recording. Any application can use these routines to provide its own script-recording interface.

For more information about the OSA event recording mechanism, see Recording OSA Events. For more information about the Recorded Text event, see Recording Scripts.

A scripting component that supports the functions described in this section has the kOSASupportsRecording bit set in the *componentFlags* field of its component description record.

Executing Scripts in One Step

The OSALoadExecute, OSACompileExecute, OSADoScript functions combine the capabilities of several other scripting component functions so that an application can execute a script in a single step. You can use these functions if you know that the script data to be executed will be executed only once.

A scripting component that supports the functions described in this section has the kOSASupportsConvenience bit set in the *componentFlags* field of its component description record.

Manipulating Dialects

Scripting components that provide several dialects may provide five functions that allow you to switch between dialects dynamically and get information about currently available dialects. The codes for specific dialects are provided by the scripting component.

The OSASetCreateProc function sets the current dialect, and the OSAGetCreateProc function gets the dialect code for the current dialect. The OSAAvailableDialectCodeList function returns a list of codes for a scripting component's dialects. You can pass any of these codes to the OSAGetDialectInfo function to get information about a specific dialect.

Instead of using the OSAAvailableDialectCodeList and OSAGetDialectInfo functions, you can use the OSAAvailableDialects function to get a descriptor list that contains information about all of the currently available dialects for a scripting component. However, it is usually more convenient to get information about just one dialect.

A scripting component that supports the functions described in this section has the kOSASupportsDialects bit set in the *componentFlags* field of its component description record.

Using Script Contexts to Handle OSA Events

The optional routines described in this section allow your application to use script contexts to handle OSA events. One way to do this is to install a general OSA event handler in your application's special handler dispatch table. The general OSA event handler provides initial handling for every OSA event received by your application. (For an example of such a handler, see Using a Script Context to Handle an OSA Event.)

The general OSA event handler extracts the event's direct parameter, obtains a script ID for the script context associated with the object described in the direct parameter, and passes the OSA event and the script ID to either OSAExecuteEvent or OSADoEvent. The main difference between these two functions is that OSAExecuteEvent returns a script ID for the resulting script value, whereas OSADoEvent returns a reply OSA event that includes either the resulting script value or information about any errors that occurred.

If the scripting component determines that a script context cannot handle the specified event (for example, if an Object REXX script context does not include statements that handle the event), OSAExecuteEvent and OSADoEvent return errAEEventNotHandled. This causes the OSA Event Manager to look for an appropriate handler in the application's OSA event dispatch table or elsewhere, using standard OSA event dispatching. If the scripting component determines that a script context passed to OSAExecuteEvent or OSADoEvent can handle the event, the function attempts to use the script context for that purpose.

Script contexts can in turn pass an event to a resume dispatch function with a statement that is equivalent to an Object REXX continue statement. The OSASetResumeDispatchProc and OSAGetResumeDispatchProc functions allow your application to set and get pointers to the resume dispatch function used by a scripting component. These functions use the following type for a pointer to a resume dispatch function:

```
TYPE AEHandlerProcPtr = EventHandlerProcPtr;
```

A resume dispatch function takes the same parameters as an OSA event handler and dispatches an event to an application's standard handler for that event.

If you need to create a new, empty script context, you can use the OSAMakeContext function.

A scripting component that supports the functions described in this section has the kOSASupportsEventHandling bit set in the c*omponentFlags* field of its component description record.

Generic Scripting Component Routines

To manipulate and execute scripts written in different scripting languages, your application can either open a connection with each corresponding scripting component individually or open a single connection with the generic scripting component. For information about how to connect with scripting components, see Connecting to a Scripting Component.

If you open a connection with the generic scripting component, it in turn attempts to open connections dynamically with the appropriate scripting component for each script that it executes or manipulates. To provide this capability, the generic scripting component must be able to determine which scripting component created any script ID passed as a parameter to a standard scripting component routine. Because different scripting components may end up using the same script ID to refer to different scripts, the generic scripting component uses its own generic script IDs. The generic scripting component translates generic scripting IDs into the corresponding component-specific script IDs and vice versa when necessary.

A generic script ID is a script ID of type GenericID.

```
TYPE GenericID = OSAID;
```

You do not need to know in detail how the generic scripting component keeps track of script IDs. However, you should be aware that the script IDs to which your application refers when it uses the generic scripting component are not the same as the script IDs used by scripting components that s actually manipulate and execute scripts.

If you are writing a script editor or recorder, you must pass the existing script ID to OSACompile or OSAStartRecording when you are recompiling or recording into an existing script. This ensures that the script is recompiled or recorded using the same scripting component that originally created the script. If instead you pass kOSANullScript to these routines, the new script is compiled or recorded using the default scripting component. Each instance of the generic scripting component has its own default scripting component. The section Getting and Setting the Default Scripting Component which follows, describes routines provided by the generic scripting component that allow you to get and set the default scripting component.

The generic scripting component supports the standard scripting component routines. However, most scripting components also support their own component-specific routines. You cannot use the generic scripting component to call a component-specific routine. Instead, you must use an instance of the specific scripting component that supports the routine.

To facilitate the use of component-specific routines, the generic scripting component allows you to identify the scripting component that created stored script data, get an instance of a specified scripting component, and convert between generic script IDs and component-specific script IDs. The section Using Component-Specific Routines describes the generic scripting component routines that allow you to perform these tasks.

Some generic scripting component routines take or return a component subtype of type ScriptingComponentSelector.

```
TYPE ScriptingComponentSelector = OSType;
```

You can use subtype codes of this type to identify specific scripting components.

Getting and Setting the Default Scripting Component

The default scripting component for any instance of the generic scripting component is initially Object REXX, but you can change it if necessary. The OSAGetDefaultScriptingComponent and OSASetDefaultScriptingComponent functions allow you to get and set the default

scripting component.

Using Component-Specific Routines

You cannot use the generic scripting component to call a component-specific routine. Instead, you must use an instance of the specific scripting component that supports the routine.

To facilitate the use of component-specific routines, the generic scripting component allows you to identify the scripting component that created stored script data, get an instance of a specified scripting component, and convert between generic script IDs and component-specific script IDs.

If you want to identify the scripting component that created a storage descriptor record but do not want to load the script, use the OSAGetScriptingComponentFromStored function. When you need to use a specific scripting component, the OSAGetScriptingComponent function allows you to get a component instance for that scripting component.

The OSAGenericToRealID and OSARealToGenericID functions allow you to convert between generic script IDs and component-specific script IDs.

Routines Used by Scripting Components

Scripting components can call three routines to manipulate the trailers for generic storage descriptor records. Writing a Scripting Component provides general guidelines for writing a scripting component.

Manipulating Trailers for Generic Storage Descriptor Records

All scripting components must use the OSAGetStorageType, OSAAddStorageType, and OSARemoveStorageType functions described in this section to add, remove, and inspect the trailers appended to script data in generic storage descriptor records.

For more information about generic storage descriptor records, see Saving Script Data.

Application-Defined Routines

Your application can provide alternative active, send, and create functions for use by scripting components during script execution. All scripting components support routines that allow you to set and get the current active function called periodically by the scripting component during script execution. Some scripting components also support routines that allow you to set and get the current send and create functions used by the scripting component when it creates and sends OSA events during script execution.

This section provides the syntax declarations for the active, send, create, and resume dispatch functions.

MyActiveProc

Your application can provide an alternative active function that performs periodic tasks during script compilation.

Syntax

```
#include <os2.h>

OSErr MyActiveProc (LongInt refCon)
```

Parameters

refCon (LongInt) - input
A reference count.

Returns

rc (OSErr) - returns
Return code.

Remarks

Every scripting component calls an active function periodically during script compilation and execution and provides routines that allow your application to set or get the pointer to the active function. If you do not set an alternative active function for a scripting component, it uses its own default active function.

MyAECreateProc

Your application can provide an alternative create function to gain control over the creation and addressing of OSA event. This can be useful, for example, if your application needs to add its own transaction code to the event. An alternative create function takes the same parameters as the AECreateOSAEvent function plus a reference constant.

Syntax

```
#include <os2.h>

OSErr MyAECreateProc (AEEventClass theAEEventClass, AEEventID theAEEventID,
                      AEAddressDesc target, Integer returnID,
                      LongInt transactionID, OSAEvent result,
                      LongInt refCon)
```

Parameters

theAEEventClass (AEEventClass) - input
The event class of the OSA event to be creaed.

theAEEventID (AEEventID) - input
The event ID of the OSA event to be created.

target (AEAddressDesc) - input
The address of the server application.

returnID (Integer) - input
The returns ID for the OSA event or kAutoGenerateReturnID if the OSA Event Manager assigns a return ID that is unique to the current process.

transactionID (LongInt) - input
The transaction ID for the OSA event. A *transaction* is a sequence of OSA events that are sent back and forth between the client and server applications, beginning with the client's initial request of a service. All OSA events that are part of a trancaction must have the same transaction ID.

This parameter can also be set to kAnyTransactionID if no transaction ID is being used.

result (OSAEvent) - input
The OSA events that are being created.

refCon (LongInt) - input
A reference constant.

Returns

rc (OSErr) - returns
Return code.

Remarks

Every scripting component calls a create function whenever it creates an OSA event during script execution and provides routines that allow you to set or get the pointer to the create function. If you do not set an alternative create function for a scripting component, it uses the standard OSA Event Manager function AECreateOSAEvent, which it calls with its own default parameters.

For descriptions of the scripting component routines you can use to set or get the pointer to a scripting component's create function, see Manipulating the Create and Send Functions.

For information about create functions, see Alternative Create Functions. For a discussion of the parameters for the AECreateOSAEvent, function, see Creating an OSA Event.

MyAESendProc

Your application can provide an alternative send function that performs almost any action instead of or in addition to sending OSA event. For example, before sending an OSA event, an alternative send function can modify the event or save a copy of the event. An alternative send function takes the same parameters as the AESend function plus a reference constant.

Syntax

```
#include <os2.h>

OSErr MyAESendProc (OSAEvent theOSAEvent, OSAEvent reply,
                    AESendMode sendMode, AESendPriority sendPriority,
                    LongInt timeOutInTicks, IdleProcPtr idleProc,
                    EventFilterProcPtr filterProc, LongInt refCon)
```

Parameters

theOSAEvent (OSAEvent) - input
The OSA event to be sent.

reply (OSAEvent) - input
The reply OSA event. This parameter is returned by the AESend method when kAEWaitReply flag is specified in the s*endMod*e parameter. If the kAEQueueReply flag is specified in the s*endMod*e parameter and the event is not going to be directly dispatched, the reply OSA event is received in the event queue. If kAENoReply flag is specified, the reply OSA event returned by this function is a null-descriptor record. If kAEWaitReply is specified in the s*endMod*e parameter, the application is responsible for using the AEDisposeDesc method to dispose of the descriptor record returned in the reply parameter.

sendMode (AESendMode) - input
A flag indicating the appropriate mode.

These mode flags are divided into five categories. The kAEAlwaysInteract, kAECanInteract, and kAENeverInteract flags are used to communicate the user interaction preference to the server application, and only one of these can be set. The kAECanSwitchLayers flag is used to set the application switch mode. The flags kAENoReply, kAEQueueReply, and kAEWaitReply specify the reply mode for an OSA event, and only one of these can be set. The last flag, kAEWantReceipt, sets the return receipt mode.

If your application is sending an event to itself, you can set the kAEDontRecord or the kAEDontExecute flag to prevent the event from being recorded or to ask the OSA Event Manager to record the event without your application actually receiving it.

The server is responsible for interrogating the following flags and honoring the client's requirements: kAENeverInterace, kAECannotInteract, kAEAlwaysInterace, and kAECanSwitchLayers.

kAEAlwaysInteract
The server application can interact with the user in response to the OSA event-by convention, whenever the server application normally asks a user to confirm a decision or interact in any other way, even if no additional information is needed from the user.

kAECanInteract
The server application can interact with the user in response to the OSA event-by convention, if the user needs to supply information to the server. This flag is the default when an OSA event is sent to a local application.

kAEDontExecute
Your application is sending an OSA event to itself for recording purposes only-that is, you want the OSA Event Manager to send a copy of the event to the recording process but you do not want your application

actually to receive the event.

kAEDontRecord
: Your application is sending an event to itself but does not want the event recorded. When OSA event recording is on, the OSA Event Manager records a copy of every event your application sends to itself except for those events for which this flag is set.

kAENeverInteract
: The server application should never interact with the user in response to the OSA event.

kAENoReply
: Your application does not want a reply OSA event; the server processes your OSA event as soon as it has the opportunity. The OSA Event Manager returns immediately from the AESend function.

kAEQueueReply
: Your application wants a reply OSA event; the reply appears in your event queue as soon as the server has the opportunity to process and respond to your OSA event. The OSA Event Manager returns immediately from the AESend function, except for direct dispatched events. Direct dispatched events cause the reply to be returned in the *reply* parameter of this function.

kAEWaitReply
: Your application wants a reply OSA event and is willing to give up the processor while waiting for the reply; for example, if the server application is on the same computer as your application, your application yields the processor to allow the server to respond to your OSA event. The OSA Event Manager creates a new thread for the calling process and blocks on this thread, waiting for the server to reply.

  The process will continue to receive all messages (events) on its message queue.

kAEWantReceipt
: The sender wants to receive a return receipt for this OSA event from the OSA Event Manager. (A return receipt means only that the receiving application accepted the OSA event; the OSA event may or may not be handled successfully after it is accepted.) If the receiving application does not send a returned receipt before the request times out, AESend returns errAETimeout as its function result.

sendPriority (AESendPriority) - input
: A flag indicating the send priority. This parameter can be set to one of the following values:

kAENormalPriority
: OSA event is posted at the end of the event queue.

kAEHighPriority
: OSA event is posted at the front of the event queue.

timeOutInTicks (LongInt) - input
: The length of time, in ticks, that the client application is willing to wait for the reply or return receipt from the server application before timing out. This parameter can also be set to one of the constants in the following list:

kAEDefaultTimeout
: The OSA Event Manager provides an appropriate time-out duration.

kAENoTimeout
: The OSA event never times out.

idleProc (IdleProcPtr) - input
: Reserved value, must be NULL.

filterProc (EventFilterProcPtr) - input
: Reserved value, must be NULL.

refCon (LongInt) - input
: A reference constant.

Returns

rc (OSErr) - returns
: Return code.

Remarks

Every scripting component calls a send function whenever it sends an OSA event during script execution and provides routines that allow you to set or get the pointer to the send function. If you do not set an alternative send function for a scripting component, it uses the standard OSA Event Manager function AESend, which it calls with its own default parameters.

For descriptions of the scripting component routines you can use to set or get the pointer to a scripting component's send function, see Manipulating the Create and Send Functions.

For more information about send functions, see Alternative Send Functions. For a description of the parameters for the AESend function, see Sending OSA Events.

MyResumeDispatch

Your application can provide a resume dispatch function that a scripting component calls during script execution to dispatch OSA event directly to an application's default handler for an OSA event. A resume dispatch function takes the same parameters as an OSA event handler.

Syntax

```
#include <os2.h>

OSError MyResumeDispatch (OSAEvent theOSAEvent, OSAEvent reply, LongInt ref
```

Parameters

theOSAEvent (OSAEvent) - input
The OSA event to be dispatched.

reply (OSAEvent) - input
The default reply OSA eventt provided by the OSA Event Manager.

refCon (LongInt) - input
The reference constant stored in the OSA event dispatch table for the OSA event.

Returns

rc (OSError) - returns
Return code.

Remarks

If a script specifies that the OSA event should be passed to an application's standard handler for that event the scripting component executing the script passes the event to the resume dispatch function currently being used by the scripting component. The resume dispatch function should dispatch the event directly to the application's standard handler for that event. If you use script contexts to handle OSA events, you may need to provide a resume dispatch function. If you can rely on standard OSA event dispatching to dispatch the event correctly, you do not need to provide a resume dispatch function. Instead, you can use the OSASetResumeDispatchProc routine to specify that the OSA Event Manager should use standard OSA event dispatching instead of a resume dispatch function.

For a discussion of the use of script contexts to handle OSA events, see Using a Script Context to Handle an OSA Event.

Summary of Scripting Components

The following table summarizes the methods related to scripting components.

Function Name	Description
OSAAddStorageType	Adds a trailer to the script data in a generic storage descriptor record.
OSAAvailableDialectCodeList	Obtains a descriptor list containing dialect codes for each of a scripting component's currently available dialects.
OSAAvailableDialects	Obtains a descriptor list containing information about each of the currently available

	dialects for a scripting component.
OSACoerceFromDesc	Obtains the script ID for a script value that corresponds to the data in a descriptor record.
OSACoerceToDesc	Coerces a script value to a descriptor record of a desired descriptor type.
OSACompile	Compiles the source data for a script and obtains a script ID for a compiled script or script context.
OSACompileExecute	Compiles and executes a script in a single step rather than calling OSACompile and OSAExecute functions.
OSACopyID	Updates script data after editing or recording and performs undo or revert operations on script data.
OSADisplay	Converts a script value to text.
OSADispose	Reclaims the memory occupied by script data.
OSADoEvent	Handles an OSA event with the aid of a script context and obtains a reply event.
OSADoScript	Handles an OSA event with the aid of a script context and obtains a reply event.
OSAExecute	Executes a compiled script or a script context.
OSAExecuteEvent	Handles an OSA event with the aid of a script context and obtains a script ID for the resulting script values.
OSAGenericToRealID	Converts a generic script ID to the corresponding component-specific script ID.
OSAGetActiveProc	Obtains a pointer to the active function that a scripting component is currently using.
OSAGetCreateProc	Obtains a pointer to the create function that a scripting component is currently using to create OSA events.
OSAGetCurrentDialect	Obtains the dialect code for the dialect currently being used by a scripting component.
OSAGetDefaultScriptingComponent	Obtains the subtype code for the default scripting component associated with an instance of the generic scripting component.

OSAGetDialectInfo	Obtains information about a specified dialect provided by a specified scripting component.
OSAGetResumeDispatchProc	Obtains the resume dispatch function currently being used by a scripting component instance during execution of an Object REXX continue statement or its equivalent.
OSAGetScriptInfo	Obtains information about script data.
OSAGetScriptingComponent	Obtains the instance of a scripting component for a specified subtype.
OSAGetScriptingComponentFromStored	Obtains the subtype code for a scripting component that created a storage descriptor record.
OSAGetSendProc	Obtains a pointer to the send function that a scripting component is currently using.
OSAGetSource	Decompiles the script data identified by a script ID and obtains the equivalent source data.
OSAGetStorageType	Retrieves the scripting component subtype from the script trailer appended to the script data in generic storage descriptor record.
OSALoad	Loads script data.
OSALoadExecute	Loads script data.
OSAMakeContext	Obtains a script ID for a new script context.
OSARealToGenericID	Converts a component-specific script ID to the corresponding generic script ID.
OSARemoveStorageType	Removes a trailer from the script data in a generic storage descriptor record.
OSAScriptError	Returns information about errors that occur during script execution.
OSAScriptingComponentName	Obtains the name of a scripting component.
OSASetActiveProc	Sets the active function that a scripting component calls periodically while executing a script.
OSASetCreateProc	Specifies a create function that a scripting component should use instead of the OSA Event Manager's AECreateOSAEvent function when creating OSA events.
OSASetCurrentDialect	Specifies a create

	function that a scripting component should use instead of the OSA Event Manager's AECreateOSAEvent function when creating OSA events.
OSASetDefaultScriptingComponent	Sets the default scripting component associated with an instance of the generic scripting component.
OSASetDefaultTarget	Sets the default target application for OSA events.
OSASetResumeDispatchProc	Sets the resume dispatch function called by a scripting component during execution of an Object REXX continue statement or its equivalent.
OSASetScriptInfo	Sets information about script data.
OSASetSendProc	Specifies a send function that a scripting component should use instead of the OSA Event Manager's AESend function when sending OSA events.
OSAStartRecording	Turns on OSA event recording and records subsequent OSA events in a compiled script.
OSAStopRecording	Turns off OSA event recording.
OSAStore	Returns a handle to the script data in the form of a storage descriptor record.
